University of West Bohemia
Faculty of applied sciences

Algorithms for manipulating large geometric data

Ing. Jiří Skála

doctoral thesis
submitted in partial fulfilment of the requirements
for a degree of Doctor of philosophy
in Computer science and engineering

Supervisor: prof. Dr. Ing. Ivana Kolingerová
Department: Department of computer science and engineering
Pilsen 2012

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Algoritmy pro manipulaci s velkými geometrickými daty

Ing. Jiří Skála

disertační práce
k získání akademického titulu doktor
v oboru Informatika a výpočetní technika

Školitelka: prof. Dr. Ing. Ivana Kolingerová
Katedra: Katedra informatiky a výpočetní techniky
Plzeň 2012

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni. Prohlašuji, že tuto práci jsem zpracoval samostatně s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této práce.

V Plzni dne 13.7.2012 Jiří Skála

Abstract

This thesis deals with manipulating huge geometric data in the field of computer graphics. The proposed approach uses a data stream technique to allow processing gigantic datasets that by far exceed the size of the main memory. The amount of data is hierarchically reduced by clustering and replacing each cluster by a representative. The input data is organised into a hierarchical structure which is stored on the hard disk. Particular clusters from various levels of the hierarchy can be loaded on demand. Such a multiresolution model provides an efficient access to the data in various resolutions.

The Delaunay triangulation (either 2D or 3D) can be constructed to introduce additional structure into the data. The triangulation of the top level of the hierarchy (the lowest resolution) constitutes a coarse model of the whole dataset. The level of detail can be locally increased in selected parts by loading data from lower levels of the hierarchy. This can be done interactively in real time. Such a dynamic triangulation is a versatile tool for visualisation and maintenance of large geometric models. Further applications include local computations, such as height field interpolation, gradient estimation, and mesh smoothing. The algorithms can take advantage of a high local detail and a coarse context around.

The method was tested on large digital elevation maps (digital terrain models) and on large laser scanned 3D objects, up to half a billion points. The data stream clustering processes roughly 4 million points per minute, which is rather slow, but it is done only once as a preprocessing. The dynamic triangulation works interactively in real time.

Title image: The statue of David by Michelangelo, various stages of digital processing. Laser scanned data kindly provided by the Digital Michelangelo project, Stanford computer graphics laboratory [132].

This dissertation was supported by the following projects:
Czech science foundation (GAČR), project 201/09/0097
Ministry of education, youth and sports, project 2C 06002 (VIRTUAL)
Ministry of education, youth and sports, project KONTAKT 5/2005-06
Ministry of education, youth and sports, project LC 06008 (CPG)

Abstrakt

Tato disertační práce se zabývá manipulací s velkými geometrickými daty v oblasti počítačové grafiky. Navržený přístup používá data stream techniku. Ta dovoluje zpracovat obrovská data, jejichž objem výrazně překračuje velikost operační paměti. Množství dat je hierarchicky redukováno clusterováním (shlukováním). Každý cluster (shluk) je nahrazen reprezentantem. Vstupní data jsou zorganizována do hierarchické struktury, která je uložena na pevný disk. Jednotlivé clustery z různých úrovní hierarchie pak mohou být nahrávány podle potřeby. Vytvořený model ve více rozlišeních poskytuje efektivní přístup k datům v různých úrovních detailů.

Pro zavedení další struktury do dat je možné zkonstruovat Delaunayho triangulaci (ve 2D nebo v 3D). Triangulace nejvyšší úrovně hie-rarchie (data v nejnižším rozlišení) představuje hrubý model celých dat. Množství detailů může být ve vybraných částech lokálně zvýšeno nahráním dat z nižších úrovní hierarchie. Lze to dělat interaktivně v reálném čase. Tato dynamická triangulace je všestranný nástroj pro vizualizaci a zacházení s velkými geometrickými modely. Další aplikace zahrnují lokální výpočty, jako např. interpolace nadmořské výšky, odhad gradientu nebo vyhlazování trojúhelníkové sítě. Algoritmy mohou využít lokálně velkou úroveň detailů a hrubý kontext v okolí.

Metoda byla testována na velkých digitálních modelech terénu (digitální výškové mapy) a na velkých laserových scanech 3D objektů až do velikosti půl miliardy bodů. Datastreamové clusterování zpracuje přibližně 4 miliony bodů za minutu, což je poměrně pomalé, ale je to provedeno jen jednou jako předzpracování. Dynamická triangulace pracuje interaktivně v reálném čase.

Titulní obrázek: Michelangelova socha Davida, různé fáze digitálního zpracování. Laserem scanovaná data laskavě poskytl projekt Digital Michelangelo, Stanford computer graphics laboratory [132].

Tato disertační práce byla podporována následujícími projekty:
Grantová agentura české republiky (GAČR), projekt 201/09/0097
Ministerstvo školství, mládeže a tělovýchovy, projekt 2C 06002 (VIRTUAL)
Ministerstvo školství, mládeže a tělovýchovy, projekt KONTAKT 5/2005-06
Ministerstvo školství, mládeže a tělovýchovy, projekt LC 06008 (CPG)

Contents

Chapter 1

Introduction

The computing power is constantly growing but so does the capability of new technologies to produce huge amounts of data. This thesis deals with large data in the field of computer graphics. The Stanford 3D Scanning Repository [132] gives a nice example of the trend. The repository offers several laser scans of real world objects. The famous Bunny scanned in 1994 consists of 36 000 points. The biggest model today is the new scan of the Michelangelo's statue of David. It was scanned in 2011 in a quarter-millimeter resolution. The data comprises 468.6 million points. There are aerial scans of the Earth surface [139], the digital elevation map (digital terrain model) is available in a resolution of 900 m per point, containing 310 million of points. Large digital models are used in construction industry. The Power Plant from the Walkthru Project [137] is made of 13 million triangles. Large data can be found in medicine. The images of human body from the Visible Human project [135] consist of about 15 GB of CT, MRI, and photo data.

It is clear that such huge data cannot be handled by ordinary algorithms. The data would not fit into the main memory and the processing would take too much time. Special techniques are necessary. This thesis deals with processing on standard desktop computers. It is a standard equipment of practically every office working with digital data. Even if a specialised laboratory had a supercomputer, there will probably be the need to provide the data to customers with an average hardware.

The so called out-of-core algorithms are being used to deal with large data. The processing is done using a big, slow external memory such as a disk array, and the algorithms strive to minimise the I/O operations, especially random access. The data stream concept is perfectly suitable for such a task. The data is read as a continuous stream allowing only successive access to the elements. The processing is done on the fly as the stream is read. The partial results are typically stored back to the hard disk. A small summary information may be kept in the main memory to append the following results

correctly, and to guide the further processing. A well designed data stream algorithm can process potentially infinite amount of data.

The data streams emerged in 1970's to monitor network traffic, where the data arrives at a high rate and rather simple statistics need to be computed. However, computer graphics often requires complicated computations with advanced data structures and dependencies within the data. Several passes over the stream might be needed to perform such complex operations. Perhaps this is the reason why data streams have not been much widespread in computer graphics so far. The applications in computational geometry are discussed in Section 3.2.

1.1 Problem definition

The input is a large unorganised set of data. The goal is to build a structure that would allow an efficient manipulation with the data, i.e., transmission, visualisation, or any further computational processing. The input should be processed as a data stream to be able to deal with really huge data. A coarse overview of the whole data should be available with the possibility to locally add details in selected areas.

1.2 The proposed solution

This thesis deals mainly with point data so the text will often refer to the data as the points. The proposed method has two stages. The points are first preprocessed by a data stream clustering. It makes a single linear pass over the data, builds a hierarchy of clusters, and stores it on the hard disk. It is a time consuming process but it is done only once. The cluster hierarchy is used in the second stage to construct a dynamic triangulation, which allows changing the level of detail interactively in real time. The following text describes the two stages in more detail.

A data stream clustering is used to handle huge data. The stream is processed in blocks that fit into the main memory. The clustering is computed within each block independently. The centre of every cluster is selected as the cluster representative. The centres are kept in another block at the so called higher level. All the remaining cluster members are put aside on the hard disk. When the block of the cluster representatives gets full, it is clustered again, and the resulting centres are passed to the next higher level block. This way, a hierarchy of clusters is built and stored on the hard disk. A detailed description is in Section 4.3. The cluster hierarchy constitutes a multiresolution model and it allows dynamically combining the data from various levels.

The triangulation of the points from the top level is constructed. This gives a very coarse model but it covers the whole dataset. This thesis uses

the Delaunay triangulation but other structure can be used as well, e.g., a surface triangular mesh. Every vertex in the triangulation represents a cluster of vertices at a lower level. These cluster representatives can be *expanded*, i.e., all the cluster members are inserted into the triangulation, to locally increase the level of detail. This can be done interactively in real time. Clusters can be later *collapsed*, i.e., replaced by the representative, to free memory for other data. Section 8 describes the algorithm in detail.

1.3 Summary of contributions

A new approach to handling huge geometric data is presented. The existing data stream clustering algorithm was modified to compute the facility location rather than k-means. The output of the original method are just the k "top level" clusters. This thesis suggests a format to store the entire hierarchy of clusters to the hard disk. The format provides an efficient access to selected parts of the data in various resolutions. The data stream clustering algorithm was thoroughly analysed with respect to the parameter settings and the input data characteristics. Enhancements were proposed to substantially speed up the computation. Supplementary contributions include the heuristics for the minimum Euclidean matching, and an implementation of a clustering algorithm on the GPU.

This thesis proposes the multiresolution Delaunay triangulation – both 2D and 3D – constructed from the hierarchy of clusters. A new method is presented for removing a number of points from the triangulation at the same time. The existing concept of using elliptical metrics for the Delaunay triangulation was revised with a more plain explanation of the mathematical background. The elliptical metrics is extended for use also in the clustering. A method for restoring the non-convex shape of a 3D object is proposed as an alternative to elaborate surface reconstruction algorithms.

The rest of this thesis is divided into two parts. Part I describes the theoretical principles related to this thesis and discusses the current state of the art in the area. Chapter 2 surveys the problems with large data in general, while Chapter 3 focuses on the data stream processing. Chapter 4 describes selected clustering algorithms, especially those suitable for large data. Chapter 5 introduces the triangulation, algorithms for triangulating large data, and methods for maintaining such a large structure. Part II describes the proposed approach for manipulating large geometric data. Chapter 6 describes the principles of the proposed solution. Chapter 7 presents the contributions to the field of clustering, Chapter 8 proposes the dynamic hierarchical triangulation. Chapter 9 concludes this thesis and suggests possible directions of the future work.

Part I

Background and state of the art

Chapter 2

Large Geometric Data

Large data may be found in many areas such as databases, sensor networks, network traffic monitoring or market statistics. It is also intensively studied in computer graphics. This chapter gives a general overview of selected methods for large geometric data acquisition and manipulation. Especially for the manipulation, there are many profoundly different approaches. This chapter mentions the fundamental techniques and gives a basic overview of their function. Following chapters focus on selected particular methods in detail. It is to be noted that this thesis deals with large data in computer graphics, especially data of a geometric character. Other areas such as video processing are not covered in the text. More details can be found in my technical report [124] which also gives additional references.

The term of *large data* has a continually evolving meaning as new technologies are discovered and brought to practise. This applies both to data acquisition and manipulation. The Stanford 3D Scanning Repository [132] offers a good example. The famous Stanford Bunny was scanned in 1994. With its 36 000 vertices it was considered quite a large model. Several Michelangelo's statues were scanned in 1999 including the Atlas with about 250 million vertices. The new scan of the David published in 2011 contains 468.6 million vertices.

Digital terrain models are another example of huge geometric data. Today, even the whole world is available in digital form [139]. The digital elevation map has a size of 1.9 GB, the resolution is about 900 m per sample. Visualisation of large detailed models is required in medicine, e.g., in The Visible Human Project [135]. The data from the year 2000 contains 58 GB of high resolution images. Further large models can be found in industry, see, e.g., The Walkthru Project [137]. The model of the Double Eagle Tanker consists of 82 million triangles. And we must not forget the film industry and computer games.

A lot of large geometric models come from scanning real-world objects. Today it is most often done using a laser scanner. The technology is called

LIDAR (LIght Detection And Ranging). It is an optical scanning technology that emits laser pulses and detects the reflected light. The principle is common with a radar with the difference that LIDAR uses light instead of radio waves. This way it is possible to capture a full 3D shape of virtually any object. It is even possible to mount such a device on an aeroplane and make a large detailed scan of Earth's surface. Satellite photography is used to record wide areas.

In some cases it is necessary to process the entire dataset; nothing is to be omitted. If the dataset is too large, the only possibility is to process it in pieces. There are generally two possible approaches to this task. The first one is to use a parallel and/or distributed computing to distribute the data among several (or several thousands) processing units. More on parallel and distributed processing of geometric data could be found, e.g., in [1, 78]. Another approach also processes the data in pieces but on a single computer. This is the first step to the so called data stream approach. Streaming algorithms are discussed in detail in Chapter 3.

The following sections summarise different approaches to handle large geometric data. The list of the methods mentioned here has no ambition of being complete. The techniques most relevant for this thesis are discussed.

2.1 Simplification

It is often not necessary to deal with all the data. Only a subset of the original data is sufficient. The amount of data is reduced to a lower resolution, allowing a more efficient processing at the cost of reduced quality. This is the technique of *simplification* which is especially used for visualisation.

The simplification involves removing insignificant detail from the model. The fundamental approach by Shroeder et al. [121] uses the technique of *vertex removal* also referred to as *vertex decimation*. It simplifies the model by successively removing a vertex and patching the resulting hole as illustrated in Figure 2.1. To avoid degeneracies it is necessary to check whether the vertex removal would not change the topology of the model. Preserving the topology is a valuable property, e.g., in medical applications.

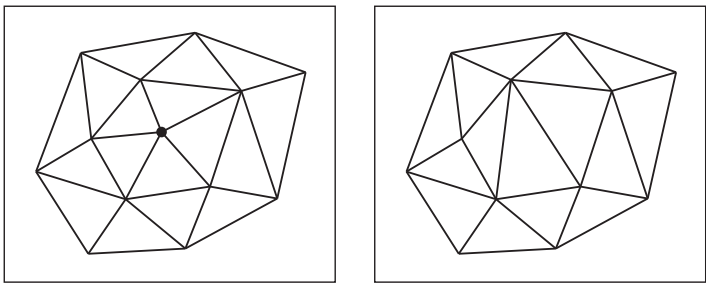


Figure 2.1: Example of a vertex removal. The marked vertex has been removed.

Another method is the *edge contraction* or *edge collapse*. It replaces two incident vertices by a single one. All the edges that were connected to both original vertices are reconnected to the new vertex. Figure 2.2 shows an example. The edge collapse is an essential part of the algorithm by Hoppe et al. [60]. They define an energy function that models the competing requirements of compact representation and geometric fidelity to the original mesh. The simplification is then solved as an optimisation problem to minimise the energy function.

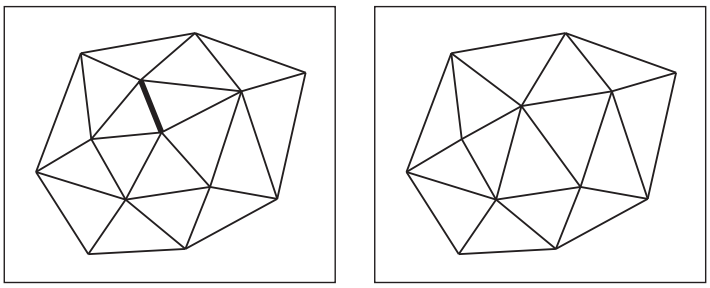


Figure 2.2: Example of an edge contraction. The marked edge has been contracted.

After the edge contraction the question is where to place the new vertex. Simple algorithms use the midpoint between the removed vertices. Advanced techniques try to minimise the error incurred. This leads to the question of error measure. Simplification algorithms need to evaluate the error of removing a particular vertex or contracting an edge, so as to decide which vertices to remove or which edges to contract. The error of removing a single vertex is often measured as a distance of the vertex from an average plane of its neighbours. For edge contraction, the distance between the involved vertices is used. Alternatively, the change in object volume may be measured. The vertices are placed into a priority queue according to their associated error which is continuously updated as the simplification proceeds. Vertices are simplified in the order of minimal error until the model is reduced to a desired size or until an error threshold is reached.

Garland and Heckbert [42] proposed an analogous method to the edge contraction – the *pair contraction*. It differs in that it can merge any two vertices, being incident or not. This way the topology may change dramatically and even independent objects may be joined together. This yields nice results when a drastic simplification is required. The algorithm uses quadric error metrics to evaluate possible contractions. The error is computed as a quadratic form which allows to compute the right replacement for the two contracted vertices and generally achieves good quality simplifications.

Cohen et al. [18] proposed a technique of *simplification envelopes*. It is a general framework within which various existing simplification algorithms can run. Simplification envelopes are a generalisation of offset surfaces.

They allow to generate mesh approximations that are guaranteed not to deviate from the original mesh by more than a pre-specified amount. Precisely speaking, all vertices of the simplified model will be within a distance ϵ from the original and vice versa. Topology is also guaranteed to be preserved. The algorithm surrounds the original mesh with two envelopes and then performs simplification within this volume. The envelopes are constructed by offsetting each vertex of the original mesh in the direction of its normal and in the opposite direction by ϵ. If should any self-intersection occur, the offset ϵ is reduced so as to avoid that. The authors present two methods for computing the envelopes as well as two simplification algorithms that can actually be used within the framework. Simplification envelopes inherently ensure that sharp edges will be preserved.

Boubekeur and Alexa [7] proposed a mesh simplification by stochastic vertex sampling. Let $M = V, T$ be the original triangle mesh. The algorithm samples a subset of k vertices $V' \subset V$. The vertex sampling is done with a special probability distribution reflecting the vertex geometric importance which is derived from the normals of the vertex and its neighbours. A topological clustering is then used to partition V into k connected components covering the whole mesh. It is done by a breadth-first traversal of the mesh from the sampled vertices V'. The triangles $T' \subset T$ with their vertices in three different components are identified. The simplified mesh is formed as $M' = V', T'$.

2.2 Level of detail

The idea of using a simpler representation of objects to improve rendering frame rate was first proposed in [17]. Level of detail techniques [91] are now often employed to render complex scenes efficiently. Not all objects present in the scene need to be rendered at full resolution. Distant objects are too small for the fine detail to be visible. Similarly, fast moving objects do not need to be rendered in high detail. Therefore simplified models are used to reduce system load. However, if an object slows down or gets close to the viewer, a more detailed model should be used. So the rendering system must be able to dynamically select a model in the appropriate resolution, thus to control the level of detail. Figure 2.3 shows a model at several different resolutions.

In earlier times, lower resolution models were prepared by hand. This had the advantage of perfect quality. The modeller person knows best which details should be preserved and which can be dropped. But the handwork is rather slow and expensive so automatic methods are used nowadays. The handmade models usually had discrete levels of detail. It means that there was the original finest model plus two or three simplified ones. Such models can be easily handled but the so called popping effect may appear when

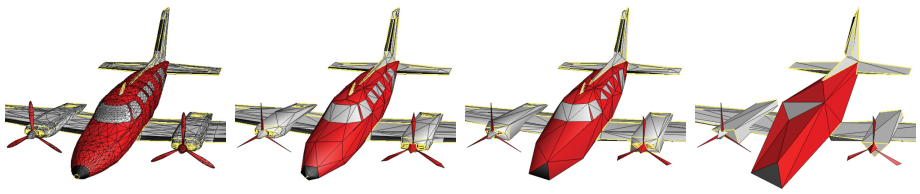


Figure 2.3: Example of a model at four different levels of detail. Adopted from [57].

switching between particular levels of detail. It means that the substitution of simplified models could be noticeable for the viewer and often even disturbing. Automatic methods [87, 120] can generate smooth transitions by gradually increasing or decreasing the amount of detail in the model. Sophisticated view-dependent methods [59, 84] can even change the detail depending on the view direction, keeping high detail only in the near parts and on the silhouette where fine details are well visible.

Most simplification algorithms can be used for the level of detail. Among them the technique of *progressive meshes* [57] is especially remarkable. The images in Figure 2.3 come from a progressive mesh. The goal is to find such a mesh M that both accurately represents the original object and has a small number of vertices. This can be expressed as a minimisation of an energy function

$$E(M) = E_{dist}(M) + E_{rep}(M) + E_{spring}(M) \tag{2.1}$$

The E_{dist} term measures the distance of the mesh from the original, the E_{rep} term is a penalisation for the number of vertices, and the E_{spring} term is to regularise the optimisation problem. Please consult [57] for more details.

The algorithm is based on [60] though it uses a modified energy function and performs only edge collapses (no edge splits or edge swaps). The key is that an edge collapse is invertible. The inverse transformation is called a vertex split and it adds a new vertex and two faces. A mesh is then stored in a much coarser version together with a sequence of vertex splits. They indicate how to incrementally refine the mesh exactly back into the original mesh. The representation thus defines a continuous sequence of meshes of increasing accuracy, from which approximations of any desired complexity can be retrieved. Moreover, *geomorphs* can be constructed between any two such meshes. A geomorph is basically a linear interpolation between two models. It makes very smooth transitions between them. Progressive meshes naturally support progressive transmission, they permit selective refinement and could be used as a mesh compression.

Hoppe [58] later introduced an extension that can do view-dependent refinement of progressive meshes by using a subsequence of vertex split operations. There is a parallel version as well [61].

Prince [112] proposed a system for generating and rendering progressive meshes for large models. His method subdivides the large triangle mesh into blocks that fit into the memory. Each block is simplified independently. Neighbouring blocks are then stitched together. The process repeats in a hierarchical fashion until all pieces have been stitched. The resulting top-level mesh is then simplified once more.

Lindstrom et al. [87] presented an algorithm for real-time level of detail reduction and rendering of polygonal surface data, especially digital terrains and other height fields. The algorithm uses a regular grid representation and employs a screen-space threshold to bound the error of the projected image.

Two steps are performed before rendering each frame. A coarse simplification is done to select discrete levels of detail for blocks of the surface mesh. A fine simplification follows in which individual vertices are considered for removal. These steps compute and generate the appropriate level of detail dynamically in real time. The method minimises the number of rendered polygons and allows smooth changes in resolution across the surface.

The fine simplification considers pairs of triangles. If the change in slope between them does not exceed a specified threshold, the triangles can be merged. The triangles are organised in a binary tree, where the smallest triangles correspond to terminal nodes, and coalesced triangles correspond to higher level nodes.

To limit the number of computations, the mesh is divided into blocks organised in a quad tree. A conservative estimate of the maximum screen-space error is computed for each block based on its bounding box. The block is eventually replaced by a lower resolution block. The mesh can be decimated by several factors before the fine simplification.

Duchaineau et al. [32] proposed the popular method known as the ROAM. It uses two priority queues to drive split and merge operations that maintain continuous triangulations built from preprocessed bintree triangles. A *triangle bintree* is a binary tree whose root is a right isosceles triangle. The children of any node are defined by splitting the parent triangle along an edge from its apex vertex to the midpoint of its base edge. Neighbours in a bintree triangulation are either from the same bintree level, or from the next or the previous level.

The split and merge operations provide a flexible framework for making updates to a triangulation. The idea is to keep priorities (error measures) for every triangle in the triangulation, starting with the coarsest triangulation, and repeatedly do a split of the highest-priority triangle. This creates a sequence of triangulations that minimise the maximum priority at every step. Adding a second priority queue allows the algorithm to start from a previous optimal triangulation when the priorities have changed, and thus take advantage of the frame to frame coherence.

Levenberg [84] introduced a method based on binary triangle trees. Instead of manipulating triangles, the method operates on clusters of geometry

called *aggregate triangles*. An aggregate triangle is a collection of triangles that substitutes for a single binary tree in a bintree triangulation. An aggregate triangle therefore corresponds to a right isosceles triangle in the grid. The terrain is divided into aggregate triangles resulting in a much shallower bintree. The geometry of aggregate triangles remains fixed for several frames so they can be efficiently cached.

Unlike recent elaborate algorithms, Lindstrom and Pascucci [88] presented a simple to implement framework for visualisation of large terrains. The emphasis is put on the layout of the data to achieve good memory coherency. They use a memory mapped file, and let the operation system take care of the paging. It is then necessary to determine a way of storing the raw data that minimises the paging. The proposed framework uses the longest edge bisection refinement [87, 32] to handle large terrains. The data is stored from coarse to fine levels of refinement in two quad trees – separately for even and odd levels. Data elements are accessed by special indexing that is derived from the quad trees. The authors pay special attention to efficient computation of the index.

2.3 Hierarchical Triangulations

Devillers [28] proposed the *Delaunay hierarchy* – a data structure for efficient point location in the Delaunay triangulation. The location structure is organised into several levels. The lowest level is the complete triangulation. Each higher level contains a triangulation of a small random sample of the level below. The point location is then done by walking in the top level triangulation to find the nearest neighbour. The walk then restarts from that neighbour at the level below.

The data structure allows points to be inserted and deleted. Deleting a point is done by simply removing it from all the levels where it appears. Inserting a point requires randomly choosing its level l, and inserting the point at every level $i, 0 \leq i \leq l$.

Bintree triangulations [32] or *hierarchies of diamonds* [87] can efficiently represent regularly sampled terrains, i.e., meshes built on a complete regular grid. Weiss and De Floriani [140] propose *Sparse Terrain Pyramids* for a compact multiresolution representation of terrains the samples of which are a subset of the regular grid. This is a way to deal with large flat areas, where the elevation is uniform, or water areas, where the elevation is unknown.

Danovaro et al. [23] developed an out-of-core multiresolution model that can be used for terrains as well as tetrahedral geological data. Their approach is based on *Multi-Tessellations* [113, 24, 114] representing a model as a base mesh at a coarse resolution, and a partially ordered set of modifications that can be applied to locally refine the base mesh. To reduce I/O operations when working with large models, the modifications are clustered

to improve efficiency. The authors developed several clustering techniques that can be divided into two classes. Sorting the modifications in the Multi-Tessellation groups the modifications that can take place one after another. Spatial grouping connects modifications that take place at the same locations in the mesh.

Multiresolution cell complexes [94] represent an object of arbitrary shape as an aggregation of simple, basic shapes, called cells. A multiresolution complex is a structure which encodes a geometric object at different resolutions. A terrain model is represented by a two-dimensional complex (a triangulation or a square grid) and a scalar function (elevation) defined piecewise on the cells, embedding the complex into three dimensions. Except visualisation, the structure supports various further operations, including point location queries, testing mutual visibility of two points, and finding contour lines.

2.4 Point based rendering

Points as rendering primitives were first discussed in [85]. The point based rendering got an increased popularity later [49, 117]. A comparison of various point based rendering techniques may be found in [118].

As the name suggests the point based rendering uses points or similar very simple primitives to render complex 3D models. Every point is defined by its coordinates and preferably has also a normal vector corresponding to the surface from where it was sampled. Optionally, the point colour may be specified. Unlike conventional triangle meshes, it works with just the points without connectivity. It does not require any information about point adjacency. So there is no need for a demanding and possibly unstable triangle mesh construction. This spares a lot of time as well as memory because there is no need to store the mesh. For large models, the problem is not only to compute the reconstruction. The problem is even to hold the whole triangle mesh in the memory.

The point based rendering displays just individual points. It is therefore necessary to render the points large enough so that they overlap and there are no gaps between them. Otherwise the rendered model would have holes in its surface. If the rendering is correct, points fill the whole surface smoothly as seen in Figure 2.4.

The most common primitives are indeed points. Do not confuse points with pixels. A point is a spot that could be rendered several pixels large. Using programmable graphics, the point size can be adjusted according to the actual projected screen size. Rendering a spot is of course much easier then rasterising a triangle, evaluating the lighting model in all three vertices and interpolating the colour. Sprites[1] are more flexible primitives in that

[1] Small 2D images integrated into a 3D scene.

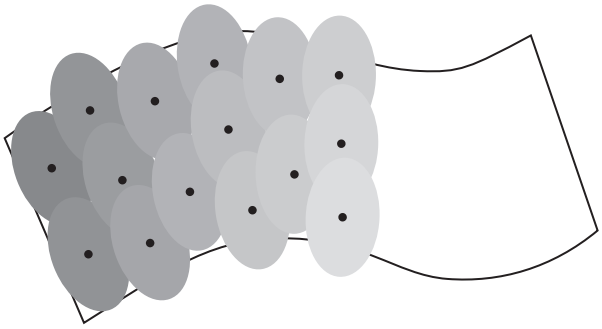


Figure 2.4: Point based rendering. Overlapping points covering a surface.

a texture can be mapped on them. The most complicated primitives are quads or triangles, still with no connectivity among them. Textures with alpha channel are mapped onto them so points can be rendered, e.g., as disks oriented according to the surface normal.

A famous point based rendering technique is the QSplat [117]. It builds a multiresolution hierarchy based on bounding spheres. Each node of the tree contains the sphere centre and radius, a normal, and the width of a normal cone. The data structure is used for culling, level of detail selection, and rendering. The algorithm recursively splits the data along the axis of the longest extend. The bounding sphere is found for each node of the tree. The node properties such as the normal and colour are computed as the average of the values in the subtree.

Rendering points is simple and allows to display even large models. But it is still profitable to employ acceleration techniques such as an octree [79]. Additionally, it allows to control the (view dependent) level of detail, do the view-frustum and the back-face culling [109]. The octree is traversed as usual. If a node lies outside of the view frustum, or if a node is determined as back-facing, or if the screen-projected size of a node is smaller than a threshold, then the sub-tree under the node is not taken into account.

Wimmer and Scheiblauer [141] introduced the method of Instant Points for rendering unprocessed point clouds. They do not make any assumptions about the density of the points or their normals. The algorithm uses nested octrees and memory optimised sequential point trees. A sequential point tree is a hierarchical structure that allows sequential processing, possibly on the GPU. Each node is associated with an error. The rendering uses the screen space error metric, i.e., it checks the projected error and either traverses the node children, or renders the node as a splat of the size to encompass the child nodes. The sequential point trees are organised into nested quadtrees to allow rendering models that do not fit into the memory.

Goswami et al. [47] proposed an out-of-core multiresolution construction and visualisation of large point data. The hierarchical level of detail is based on a multi-way kD-tree. That is a kD-tree whose nodes have more than two

children. Constructing the tree involves a top-down subdivision followed by a bottom-up simplification. The subdivision splits each tree node into a constant number of children along the axis of the greatest extent. The simplification chooses several representatives for each tree node. A grid is laid over the tree cell and the points within the grid cell are grouped according to their normal deviation, and eventually colour deviation. The representatives are computed by averaging the points in the groups.

2.5 Vertex clustering

Rossignac and Borrel [116] proposed a simplification method that uses vertex clustering to render complex scenes. Perhaps a more precise expression would be vertex quantisation. The scene is uniformly divided by a grid. The grid resolution controls the amount of simplification. Vertices falling into one grid cell form a cluster which is then replaced by a single representative vertex. This could be either the centre of mass of the cluster or the most significant vertex.

Most vertex clustering methods do such simple clustering as using the regular grid, an adaptive grid, or rounding the vertex coordinates (quantisation). The methods often require the whole, full resolution triangular mesh at the beginning. This can pose a problem if the data is large.

Luebke and Erikson [90] proposed a hierarchical dynamic simplification. It is a similar but more advanced approach. It also uses vertex clustering (but can use any other simplification algorithm) to create the so called *vertex tree*. The tree defines how vertices (possibly from different objects) will be merged together to simplify the scene. Each node in the tree contains one or more vertices. The algorithm may collapse all of the vertices within a node into a single representative vertex. Triangles whose corners have been collapsed together are removed. Likewise, a node may be expanded by replacing the representative vertex by the original node's vertices. The triangles that were removed become visible again. The vertex tree is queried dynamically at run time to generate the desired degree of simplification. The algorithm uses a screen space error metric that measures the error in pixels and therefore perfectly reflects the distortion that is actually visible.

Schaefer and Warren [119] described an out-of-core mesh simplification based on clustering the vertices using an octree. The input data must be sorted. The octree is constructed progressively until the allocated memory is exhausted. Since the data is sorted, there are always nodes where no additional vertices will be inserted. The algorithm collapses certain subtrees under such nodes to free space for further vertices, and continues building the octree.

Chapter 3

Data Streams

Data stream algorithms emerged in the last few years starting perhaps in 1970's. A brief history could be found, e.g., in [105]. First streaming applications were concerned with sorting and searching. Nowadays, data stream algorithms are used in many areas for complex analyses of massive data. Typical tasks include detecting outliers, extreme events, intrusions, track trends and perform analyses. Data streams are often found in connection with computer networks [75, 82, 142], namely monitoring the traffic or computing statistics of browser clicks and user queries. More applications include astronomical [40], satellite and meteorological surveys, financial observations [40] such as stock exchange and currency trades. An increasing interest emerges also in computer graphics [62], particularly in computational geometry [38, 110, 66, 70, 67].

This chapter introduces the fundamental techniques frequently used in data stream processing. It focuses on data streams in computer graphics. A more detailed description of the data stream concepts can be found in my technical report [124]. Muthukrishnan published a nice survey of data streams [105].

3.1 Data stream fundamentals

From a general point of view, data stream is a linear sequence of data. Every application field has different specifications, expectations, and limitations posed on the stream. There are essentially three challenges that may be concerned – to transmit the data to the program, to compute sophisticated functions on large pieces of input in an acceptable time, and to store the presented information long-term. Either of the tasks may be demanded, or all of them.

The amount of data in the stream is supposed to be extremely large so it is hard to store it or to compute complex functions by conventional algorithms. Let N be the length of the stream, i.e., the number of distinct

pieces of information, which is often supposed to be known. Data stream algorithms are usually allowed to use $\mathcal{O}(N^a)$, $a < 1$, or $\mathcal{O}(\log N)$ memory.

The data stream can be stored on hard drives or tapes. This is the offline variant [70]. Processing time is not critical and it is possible, though discouraged, to make multiple passes over the data. Random access is prohibited due to severely lowering the performance. Offline data streams allow computing complex analyses on them. This is a common case in computer graphics. By contrast, in online data streams [48] data arrive at a very high rate so it is hard and usually impossible to process it exactly. Approximate algorithms must be used. This is a common scenario in computer networks.

Many data stream algorithms are based on one of several fundamental techniques. *Shedding* is the simplest, often used on very fast and massive online data streams. It just blindly samples the data at a constant frequency. True *sampling*, as it is used in data streams, carefully selects which data to retain and which to drop [14, 48, 45, 96].

Advanced techniques such as *sticky sampling* and *lossy counting* [96] are used, e.g., to determine how frequently particular elements occur in the stream. Sticky sampling draws a sample set of distinct elements and counts how many times each of them appeared. The sample set is continuously refreshed to drop sporadic elements and to include new ones appearing in the stream. Lossy counting is a bucketing technique, i.e., it divides the stream into pieces (buckets) and processes them one after another. It counts all distinct elements in a bucket. Before proceeding to the next one, all the counters are decremented and elements whose counter reached zero are dropped. The *sketching* technique [46, 11, 19, 20, 21, 65] computes summary information on the stream. It allows various queries to be approximately answered very quickly.

3.2 Data streams in computational geometry

The streaming approach has not been used much in computer graphics so far. With increasing size of geometric models, data streams are getting into consideration in computational geometry [64]. Isenburg et al. [68] study such streaming algorithms intensively. The methods for processing geometry very often rely on the fact that the data is approximately sorted. Otherwise, some external pre-sorting must be done. This section describes a wider area of the relevant applications. The methods closely related to this thesis are described in their respective chapters, namely the data stream clustering in Section 4.3, and the streaming triangulation in Section 5.3.

3.2.1 Stream-processing points

A framework for stream-processing points was proposed by Pajarola [110] and later extended in [6]. The input is a point cloud sorted along one direc-

tion. The algorithm then sweeps the data along that direction and processes them as a stream. Points are sequentially loaded into the memory where *stream operators* (see bellow) are applied to them. Many operations such as normal computation cannot be performed on isolated points. Therefore a *working set* is maintained where points accumulate and their local neighbourhood is available. As soon as a point cannot contribute any more to any operation, it is released from the working set and written to the output. Eventually, a buffer and a deferred write may be used to preserve the global stream ordering.

Pajarola introduces *stream operators* as functions computed on a single point using only a local neighbourhood. They are applied to the points that have all necessary neighbours present in the working set. Pajarola introduces several elementary operators that are particularly important for processing raw point clouds. Except the elementary I/O operators, they are k nearest neighbours, normal computation, curvature estimation, splat size estimation (for the point based rendering see Section 2.4), and fairing (smoothing). The power of the stream-processing framework consists in that the stream operators can be concatenated and thus a complex computation on the data can be performed in a single pass. The operators are implemented as modules with input and output buffers. A module starts computing when it has enough data in the input buffer. Processed points are passed to the output buffer. This way modules hand over the data efficiently as the computation proceeds.

3.2.2 Streaming triangulations

Geometric models are usually stored in the common format comprising of a list of vertices followed by a list of triangles. There is no ordering required so a triangle may reference vertices from both ends of the vertex list, as well as a vertex may be referenced by triangles from both ends of the triangle list. It is especially notable for instance at the Stanford bunny [132] where both triangles and vertices are heavily scattered. This bad topological coherency is not particularly suitable for huge gigabyte-sized models since any processing algorithm would need random access to either of the lists. It prohibits the stream processing and makes cashing inefficient.

Isenburg and Lindstrom proposed *streaming meshes* [66] to overcome this problem. The idea is to interleave vertices and triangles so that some algorithm processing the mesh will come to the right vertices when they are needed. Vertices that will not be needed any more are *finalised*, i.e., marked that they can be freed from memory. An algorithm processing an interleaved mesh can thus hold just the vertices that are actually needed. It uses little memory and does not need random access to the whole mesh.

The authors suggest two variants of streaming meshes. The pre-order format introduces a vertex just before the triangle that references it for the

first time. The vertex is finalised as soon as it is referenced by the last incident triangle. The post-order format first introduces triangles. As soon as all triangles incident to a vertex are present, the vertex is introduced and implicitly immediately finalised.

Many mesh generating applications work in such a way that the output can be easily stored in the streaming format. Authors also propose methods how to reorganise existing models. Isenburg at al. later introduced compression algorithms for triangular and tetrahedral meshes [69, 67]. They both use the streaming mesh format so the compressor only holds the vertices that are relevant to the simplices currently being compressed.

Denker et al. [26] proposed a real time triangulation of a point stream generated by a laser scanner. The data points are reduced by clustering them into the so called *neighbourhood balls*. The balls are subsets of vertices with similar positions. The radius of each ball depends on the local point density and the estimated curvature. Subsequently only the averages from the neighbourhood balls are used as vertices in the triangulation.

The neighbourhood balls serve three purposes. First, they are used to collect data points lying close together to reduce the number of vertices in the triangulation. Second, the local oriented surface normal is computed from the points in every ball. Third, averaging the points gives the position of the vertex for the triangulation. The neighbourhood balls are organised in an octree for efficient geometric neighbourhood searches.

When a neighbourhood ball is added or changed, the triangulation is updated locally. The vertices V_i potentially affected by the ball change are collected and projected on the estimated tangent plane. The border polygon of V_i is adjusted to match the current triangulation. The local triangulation of V_i is constructed and inserted into the current triangulation.

Chapter 4

Clustering

Over the years, clustering has evolved into a really general concept. It is used in data analysis [31, 3], data mining [37] (specially mining large databases [106, 146]), information retrieval [115], image segmentation [71], pattern recognition [4] and other scientific fields [72, 73]. It can also be found in non-technical disciplines such as archeology or marketing.

The principle of clustering is grouping similar elements together. Depending on the area of application, elements could be anything – from points in 1D space[1] to 3D objects, entire images, documents or database entries. Each element is described by a characteristic vector. Points are specified by their spatial coordinates. Some abstraction is needed for more complex elements to express their properties in numbers and to get a suitable representation. The majority of clustering tasks are NP-hard problems so most algorithms produce only approximate results. This thesis focuses on clustering of points in 3D, although higher dimensional points with additional properties will be studied as well. Presented algorithms were successfully used also for clustering triangles.

This chapter briefly introduces the main concepts of clustering. It focuses on the facility location problem and on the clustering algorithms suitable for large data. A broader survey can be found in my technical report [124] along with references to even more detailed papers. A comprehensive overview of clustering can be found, e.g., in [33, 55, 31, 72, 73, 97].

4.1 Distance measures

A distance measure is required to compute the similarity between elements. By far the most common is the Euclidean distance. Given two elements $\mathbf{x}_i = \{x_{i1}, x_{i2}, \dots x_{iD}\}$, $\mathbf{x}_j = \{x_{j1}, x_{j2}, \dots x_{jD}\}$ in D dimensional space, the

[1] Clustering the depth information for rendering.

Euclidean distance is defined as

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2} \tag{4.1}$$

It works well for compact spherical clusters. Different measures are used for special applications, e.g., to detect clusters of a complex shape or for pattern matching. The Minkowski metric is a generalisation of the Euclidean distance and is defined as

$$d_p(\mathbf{x}_i, \mathbf{x}_j) = \sqrt[p]{(\mathbf{x}_i - \mathbf{x}_j)^p} \tag{4.2}$$

The Euclidean distance is assumed throughout this thesis, unless otherwise noted. It is the most common measure, very simple, and works well in many scenarios.

4.1.1 Anisotropy

Non-spherical clusters may be appropriate in special cases. There are the so called anisotropic materials that have variable properties in different directions. A nice example is wood – it is very strong along the grain, but transversely, it breaks easily. Anisotropy can be often encountered in geology. Seismic anisotropy is the variation of seismic wavespeed with direction. Geological formations with distinct layers of sedimentary material can exhibit anisotropy in electrical conductivity. This property is used in the gas and oil exploration industry. The hydraulic conductivity of water sources is often anisotropic. This is necessary to take into account when calculating groundwater flow. Most common rock forming minerals are anisotropic, including quartz and feldspar. Anisotropy in minerals is most reliably seen in their optical properties such as the birefringence of calcium crystal. Medical ultrasound imaging uses the fact that soft tissues have different echo depending on the angle of the sound source. Computer graphics may be interested in anisotropic surfaces, such as velvet, that change their appearance when rotated about their normal. Anisotropic metrics are useful to grasp such special problems.

This thesis studies the *elliptical metric* in detail because it is relatively simple and yet provides enough flexibility. An elliptical metric can be viewed as an elliptical elongation of the classical Euclidean space. It allows the clusters to have an elliptical shape rather than spherical. Ellipsoids can significantly better match phenomena such as those mentioned in the previous paragraph. The effect on a Delaunay triangulation is that edges tend to go along the direction specified by the ellipse. Thus they can be adapted to, e.g., curvature directions on a terrain surface. The edges will be aligned with a mountain ridge or root and hence better fit terrain breaks. The triangulation can be prepared for later deformations so as to avoid degenerate cases.

The following two subsections explain how to work with the elliptical metric. The fundamental idea is to transform all the points into the Euclidean space and then work with them as usual. This is well suitable, e.g., for constructing the Delaunay triangulation which requires complex computations. The transformation is described in Section 4.1.2. Some applications such as the clustering only require to measure the distances. For this purpose it is possible to compute the so called elliptical distance directly without the transformation. This is described in Section 4.1.3.

The following text will occasionally mention the Delaunay triangulation which is described later in Section 5. There is no need to know the triangulation in order to understand the elliptical metrics. The parts concerning the Delaunay triangulation are included here to keep all the equations in one place.

4.1.2 Computing with the elliptical metric

Let us define the *radius* of an ellipse to be the line connecting the centre of the ellipse and any point on the ellipse. The elliptical metric is defined by an ellipse so that, using the elliptical metric, any radius of the defining ellipse has a unit length. In other words, using the elliptical metric, the defining ellipse appears to be a unit circle. The ellipse is defined by its major and minor axes. The position in space is irrelevant.

The simplest way to use the elliptical metric is to transform the points from the space with the elliptical metric into the Euclidean space, and then work with them as normal. The transformation consists of a scaling and a rotation. The idea of computing the Delaunay triangulation using elliptical metrics was proposed by Vigo and Pla [134]. They start from the matrix representation of ellipse, and so they compute the scaling differently. It does not matter when the Delaunay triangulation is computed because the in-circle tests are scale invariant. A small contribution of this thesis is the proper derivation and discussion of the mathematics presented in [134]. The equations presented here slightly differ from [134] so as to keep the scale correct which is especially important for the clustering. The following is an intuitive derivation of the equations used for computing with an elliptical metric. For the sake of simplicity, the description is in 2D space. The situation in higher dimensions is analogous.

We are looking for a linear mapping $\mathbf{M}$ that maps the defining ellipse onto a unit circle. This is done by a rotation $\mathbf{R}$ (to align the ellipse axes with coordinate axes) followed by a non-uniform scaling $\mathbf{S}$ (to scale the ellipse to a unit circle). Note that no translation is needed since the defining ellipse is centred at the origin. Finally a reverse rotation $\mathbf{R}^{\mathrm{T}}$ should be applied. It has no effect on point distances, however, we do the rotation because the transformation matrix will then be symmetric which has advantages in algebraic calculations. Please refer, e.g., to [56] for more details.

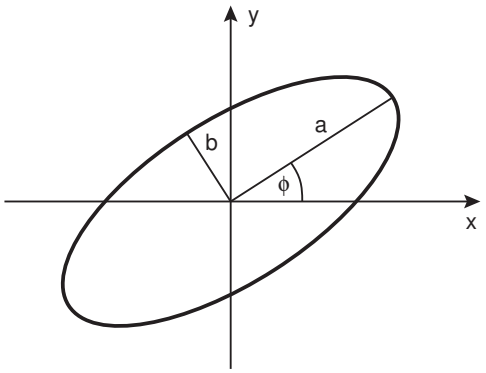


Figure 4.1: Ellipse defining the metric

Figure 4.1 shows a defining ellipse as a reference for the following equations. Using column vector notation, the point p is transformed as

$$p' = \mathbf{M}p. \tag{4.3}$$

We construct the transformation matrix $\mathbf{M}$ as

$$\mathbf{M} = \mathbf{R}^{\mathrm{T}}\mathbf{S}\mathbf{R} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \frac{1}{a} & 0 \\ 0 & \frac{1}{b} \end{pmatrix} \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \tag{4.4}$$

$$\mathbf{M} = \begin{pmatrix} \frac{1}{a}\cos^2\phi + \frac{1}{b}\sin^2\phi & \left(\frac{1}{b} - \frac{1}{a}\right)\sin\phi\cos\phi \\ \left(\frac{1}{b} - \frac{1}{a}\right)\sin\phi\cos\phi & \frac{1}{a}\sin^2\phi + \frac{1}{b}\cos^2\phi \end{pmatrix}. \tag{4.5}$$

The difference from [134] is in the scaling. They start from the matrix representation of ellipse and so they compute the scaling as

$$\mathbf{S}_{Vigo} = \begin{pmatrix} b & 0 \\ 0 & a \end{pmatrix} \tag{4.6}$$

which also results in a circle, but scaled by a factor of ab. As mentioned earlier, this does not matter when triangulations are computed.

To construct the Delaunay triangulation with respect to the elliptical metric, the input points are first transformed using Equation 4.3. The standard Delaunay triangulation of the transformed points is constructed. The same triangulation topology is then used on the original points.

4.1.3 Computing the elliptical distance

There is no need to transform the points when computing the clustering because we only need to measure the distances. We will now derive how to compute the *elliptical distance* directly without transforming the points. Let $|pq|_E$ denote the elliptical distance between points p, q. Let p', q' be the points p, q transformed into the Euclidean space. We can write

$$|pq|_E = |p'q'| = \sqrt{(p' - q')^{\mathrm{T}}(p' - q')} = \sqrt{(\mathbf{M}p - \mathbf{M}q)^{\mathrm{T}}(\mathbf{M}p - \mathbf{M}q)}. \tag{4.7}$$

Using linear algebra rules, we rearrange the equation as

$$|pq|_E = \sqrt{[\mathbf{M}(p-q)]^{\mathrm{T}}\mathbf{M}(p-q)} = \sqrt{(p-q)^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}\mathbf{M}(p-q)}. \tag{4.8}$$

Now we utilise the fact that the matrix $\mathbf{M}$ is symmetric. The elliptical distance is computed as

$$|pq|_E = \sqrt{(p-q)^{\mathrm{T}}\mathbf{M}^2(p-q)}. \tag{4.9}$$

If we look back on how the matrix $\mathbf{M}$ was constructed we find out that $\mathbf{M}^2$ can be computed as $\mathbf{R}^{\mathrm{T}}\mathbf{S}_{sqr}\mathbf{R}$ where $\mathbf{S}_{sqr}$ is the matrix $\mathbf{S}$ with its components squared.

$$\mathbf{M}^2 = \mathbf{R}^{\mathrm{T}}\mathbf{S}_{sqr}\mathbf{R} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \frac{1}{a^2} & 0 \\ 0 & \frac{1}{b^2} \end{pmatrix} \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \tag{4.10}$$

The elliptical distance can be computed directly by evaluating Equation 4.9.

4.2 Clustering approaches

A large amount of clustering techniques exist. This section briefly surveys the different approaches. The most important algorithms are described. A more detailed overview can be found in my technical report [124], further information can be found, e.g., in [73, 97].

Clustering algorithms can be divided, according to some particular principle of their function, into two opposed categories. Following paragraphs list several such divisions.

Clustering algorithms can be either *partitional* or *hierarchical*. Partitional approaches divide the data into an exact number of clusters (partitions). Hierarchical algorithms create a hierarchy of small clusters grouped into larger clusters forming a tree structure called dendrogram. The degree of clustering can be than controlled by going up and down in the hierarchy.

Another possible division is to *agglomerative* and *divisive* or *partitional* algorithms. An agglomerative approach starts with each element in a single cluster. These are then successively merged according to a similarity measure until a stopping condition is met. The algorithm usually stops when a desired number of clusters has been reached, or when there is such a low similarity between existing clusters that no further can be merged. A divisive approach works the opposite way. It starts with all the data in one large cluster. It is then repeatedly split according to a dissimilarity measure. Again the algorithm stops when there is a predetermined number of clusters or when existing clusters are homogeneous enough so that no further splits are necessary.

The clustering can be either *hard* or *fuzzy*. Hard clustering assigns each element into exactly one cluster. Fuzzy clustering determines for each element a degree of membership in several clusters.

Clustering algorithms can be *deterministic* or *stochastic*. Stochastic techniques usually use randomised algorithms. They usually run rather fast and thus are perfectly suitable for processing large amounts of data.

The clustering method may process the data all at once or work incrementally. If the algorithm works with all the data together it can in principle produce more precise results. Incremental algorithms [52] can be faster and have much lower memory requirements which is again a great benefit when processing large data. The low memory requirements come from the fact that the algorithm does not need to store all information about the data processed so far. It is only necessary to hold summary information about particular clusters which can be orders of magnitude smaller than all the data contained in them.

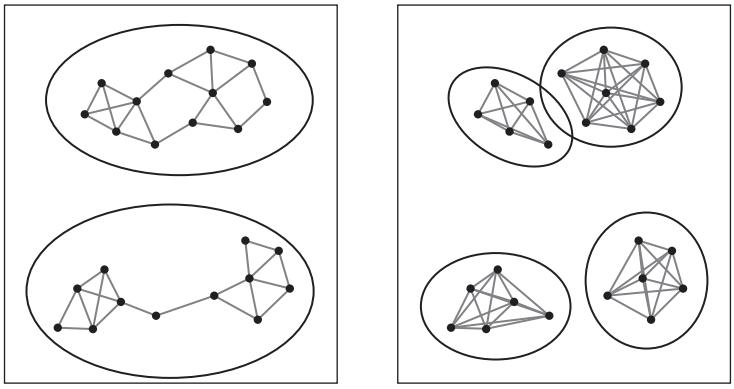
There are two major cluster representations – *centroid based* and *sample based*. The centroid based approach stores just the centre of each cluster and possibly the number of elements contained within. This is a very compact representation and sufficient for many applications. The sample based approach stores several well chosen representatives for each cluster, e.g., points at the border of the cluster. This requires more memory but also gives more information in particular about the cluster shape and possibly about how elements are distributed within the cluster.

The following sections describe selected clustering algorithms in detail. Namely two examples of the hierarchical agglomerative clustering – the single-link and the complete-link. Then the famous k-means algorithm and finally the facility location.

4.2.1 Single-link and complete-link

Single-link [131] and complete-link [76] are hierarchical agglomerative clustering approaches. They are non-incremental, deterministic and do the hard clustering. They differ in the way they measure the similarity between clusters. The single-link algorithm defines the distance between two clusters as the *minimum* pairwise distance between the elements of the two clusters, i.e., the similarity of the clusters is measured as the similarity of their *most similar* members. By contrast, the complete-link algorithm uses the *maximum* pairwise distance, i.e., the similarity of the clusters is measured as the similarity of their *most dissimilar* members.

The single-link algorithm is more versatile but tends to produce scattered or elongated clusters. This could be unpleasant but in some scenarios it is very useful to detect non-spherical clusters. The complete-link algorithm produces compact clusters. Which algorithm works better depends on the nature of input data. Figure 4.2 shows an example of both the approaches.



(a) single-link (b) complete-link

Figure 4.2: An example of the single-link and the complete-link approach

A few terms from graph theory will be necessary for the following text. A *connected component* of a graph is a maximal set of points such that there is a path connecting every pair. A *clique* in a graph is a set of points that are all connected to each other.

The single-link algorithm can be summarised as follows:

1. Start with each element in a distinct cluster. Compute distances between all the pairs of the elements.

2. Take these distances in an ascending order. For each such distance d, make a graph where all the pairs of elements closer than d are connected by an edge. When all the elements form a connected graph, stop.

3. The result is a hierarchy of graphs where an arbitrary similarity level can be selected. The clusters are the connected components of the appropriate graph.

The complete-link algorithm works the same way. The difference is that the second phase is terminated when all the elements form a single clique. When a graph is selected from the hierarchy, clusters are the maximal cliques of the graph.

The time complexity of both the algorithms is $\mathcal{O}(N^2 \log N)$, whereas the single-link can be improved to $\mathcal{O}(N^2)$. Nevertheless, both algorithms must compute N^2 distances which is the most demanding part of the computation. More on the complexity analysis can be found in [98].

4.2.2 k-means

The k-means [93] is for sure the best known clustering algorithm. It is popular for its simplicity, short running time and low memory requirements. It is a partitional non-incremental stochastic algorithm. It makes a hard clustering but there is a modification – the fuzzy c-means [5] that makes a fuzzy clustering.

The k-means algorithm works as follows:

1. Choose k cluster centres either randomly or based on some heuristics.
2. Assign each element to the closest cluster centre.
3. Recompute cluster centres as centroids of particular clusters.
4. While a convergence criterion is not met, go to step 2.

Typical convergence criteria are no (or minimal) reassignments between clusters or a low decrease in the squared error (i.e., a sum of squared distances to the cluster centres). The algorithm has a time complexity of $\mathcal{O}(NkI)$ [99], where I is the number of iterations. Practical experience shows that far less than N iterations are necessary to achieve convergence.

A significant disadvantage of the k-means is that the number of clusters k must be determined prior to starting the algorithm. It could be solved by running the algorithm several times with different settings and choosing the best result (the minimal squared error). But this considerably increases the running time. Another problem is that for some bad initial configuration the algorithm may converge to a local optimum. There are heuristics [99] for choosing the initial cluster centres so that the global optimum is reached with high probability.

A common modification of the k-means is the k-median algorithm. It has the restriction that cluster centres can only be chosen among the input elements.

4.2.3 Facility location

The facility location problem is a special clustering task. The general formulation is as follows. Let F be the set of so called *facilities* and C be the set of *clients* (in some literature expressed as D as for *demand nodes*). Every client should be serviced by (connected to) a facility. The problem is to determine which facilities to *open* and which clients should they service. The *facility cost* fc must be paid for opening a facility. The *service cost* must be paid for connecting clients to facilities (mostly based on the distance). The problem has a direct real life application. Imagine there are some cities that need to be supplied with electricity and there are several potential locations where a power plant could be built. Building a power

plant everywhere would be too expensive; as well as connecting all the cities to a single one. It is to be determined where to build a power plant and where to connect particular cities. It is necessary to find such a balance to minimise the overall costs.

Expressing the problem in a mathematical way, the task is to minimise the overall clustering cost Q defined as

$$Q = \sum_{\substack{j \in F \\ j \text{ is open}}} fc + \sum_{i \in C} c_{ij} \tag{4.11}$$

where c_{ij} is the distance between a client $i \in C$ and its facility $j \in F$. Distances are generally considered non-negative, symmetric and satisfying the triangle inequality. It is to be noted that generally there are no restrictions on the set of facilities F. It can be independent of C, a subset of C, or equal to C. There are some specialisations of the facility location problem. Facilities may have different facility costs and may have limited capacities to service just a certain number of clients. These specialisations are not considered in this thesis.

To compute an ordinary clustering of a set P, simply set $C = P$ and $F = P$. Unlike the k-means algorithm, there is no need to specify the number of clusters in advance. It is, however, necessary to choose the facility cost. It determines how the clustering will look like. A high value will produce a clustering with a low number of large clusters. Facilities are expensive so only a few of them will be opened and a lot of clients connected to each of them. On the other hand, a low facility cost will result in many small clusters. Facilities are cheap so a lot of them will be opened and clients distributed among them. Recommendations on how to set the facility cost can be found in Section 7.1.3 on page 62.

Following sections describe three different approaches to solve the facility location problem. A brief overview of them may be found in [122].

Linear programming rounding

The method was introduced by Shmoys et al. [123] based on the work by Lin and Vitter [86]. It was later extended and improved in [50, 15]. The approach is based on a linear programming relaxation. Let $x_{ij} = 1$ if the client i is connected to the facility j; $x_{ij} = 0$ otherwise. Let $y_j = 1$ if the facility j is open; $y_j = 0$ otherwise. The facility location problem can be then formulated as an integer program

$$\text{minimise} \quad \sum_{i \in C} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} fc \cdot y_j \tag{4.12}$$

subject to

$$\sum_{i \in C} x_{ij} = 1 \qquad \forall j \in F \tag{4.13}$$

$$x_{ij} \leq y_j \qquad \forall i \in C, j \in F \tag{4.14}$$

$$x_{ij} \in \{0; 1\} \qquad \forall i \in C, j \in F \tag{4.15}$$

$$y_j \in \{0; 1\} \qquad \forall j \in F \tag{4.16}$$

where 4.13 ensures that each point will be connected to exactly one facility and 4.14 ensures that it will be connected to an open facility. If we let x_{ij} and y_j be any positive real numbers, we get a linear relaxation of the integer program. This can be solved in polynomial time. The fractional solution is then rounded to the integer solution while increasing the clustering cost by a small constant factor. The proof may be found in [123].

Primal-dual algorithm

There is a related approach also based on linear programming. It was introduced by Jain and Vazirani [74] and later addressed by Charikar and Guha [12] and Mahdian et al. [95]. The method again starts with an integer program stated by Equations 4.12–4.16 and its linear relaxation. A dual program is then constructed as

$$\text{maximise} \quad \sum_{i \in C} \alpha_i \tag{4.17}$$

subject to

$$\alpha_i - \beta_{ij} \leq c_{ij} \qquad \forall i \in C, j \in F \tag{4.18}$$

$$\sum_{i \in C} \beta_{ij} \leq fc \qquad \forall j \in F \tag{4.19}$$

$$\alpha_i \geq 0 \qquad \forall i \in C \tag{4.20}$$

$$\beta_{ij} \geq 0 \qquad \forall i \in C, j \in F \tag{4.21}$$

The solution of this dual linear program gives the solution to the original problem. There is an intuitive interpretation of the dual variables α_i and β_{ij}. The α_i can be understood as a total contribution of the client i towards opening some facility and connecting the client to it. This can be divided to c_{ij} for connecting i to the facility j, and β_{ij} for opening the facility; see Equation 4.18. Equation 4.19 describes how several clients pay for opening a facility.

Based on the dual linear program and the interpretation of dual variables, an algorithm can be formulated. The original primal-dual algorithm was proposed in [74]. The following description is, however, based on [95] which is a slightly easier and more intuitive modification. A notion of time is

introduced. The algorithm starts at the time 0. Initially, all clients are unconnected, all facilities are closed and $\alpha_i = 0 \; \forall i \in C$. While $C \neq \emptyset$, for every client $i \in C$, increase the parameter α_i simultaneously at a unit rate (say by 1 in a unit time), until one of the following events occurs.

1. For an unconnected client i and an open facility j, the equality $\alpha_i = c_{ij}$ comes true. In this case, connect the client i to the facility j and remove i from C.

2. For a closed facility j, $\sum_{i \in C} \beta_{ij} = \sum_{i \in C} \max(0, \alpha_i - c_{ij}) = fc$. This means that the total contribution of clients is sufficient to open the facility j. In this case, open the facility j and for each unconnected client i for which $\alpha_i \geq c_{ij}$, connect i to j and remove i from C.

If more events occur at the same time, they can be processed in an arbitrary order.

Local search algorithm

From the general point of view, local search techniques operate on a graph on the space of all feasible solutions. Two solutions are connected by an edge if one solution can be obtained from the other by a particular type of modification. The local search technique walks in the graph along the nodes with decreasing costs, and searches for a local optimum. That is such a node whose cost is not greater than the cost of each of its neighbours. Korupolu et al. [80] analysed clustering techniques based on the local search. One of the first such techniques was proposed by Charikar and Guha [12]. First, a coarse initial solution is generated. It is then iteratively refined by a series of local search improvements. A single local search step can be briefly described as follows. A facility is chosen at random and it is determined whether opening it can improve the solution. If so, nearby clients are reassigned to the new facility. Facilities with a low number of remaining clients are then closed and their clients are reassigned to the new facility too.

Describing the local search algorithm more precisely, a facility $j \in F$ is selected at random (does not matter whether it is open or closed) and it is determined whether it can improve the current solution: If j is not already open, the facility cost would have to be paid for opening it. Next, some clients may be closer to j then to their current facility. All such clients can be reassigned to j, decreasing the connection cost. The time complexity of this first phase of one local search step is $\mathcal{O}(N)$, where N is the number of all points (we compute the distance of the facility candidate to all other points).

After that, some facilities may have just a few clients remaining. If those clients would be reassigned somewhere else, the facilities could be closed and their facility costs spared. To limit computational complexity,

reassignments are allowed only to the facility j which is being investigated. The reassignments will indeed increase connection costs, but the savings for closing the facilities (sparing their facility costs) could be larger. This second phase has $\mathcal{O}(N)$ time complexity too. It is computed simultaneously with the first phase in practise.

Figure 4.3 illustrates one local search improvement. Facilities are shown as big circles. Lines show the assignment of points to facilities. The original situation is on the left. The big grey circle denotes the facility candidate. Gray dashed lines show prepared reassignments. The situation after reassignments and closures is shown on the right. You can see that the candidate facility was actually opened. Some points were reassigned to it and the facility at the bottom was closed.

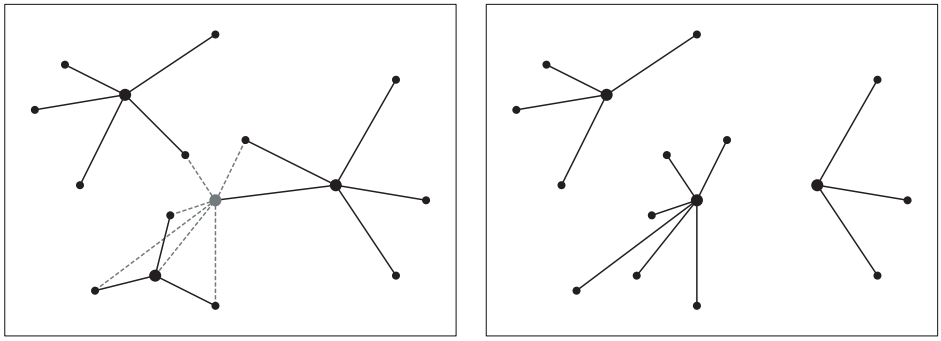


Figure 4.3: Situation before and after one local search step

The possible improvement of the current solution is computed by the *gain* function. If $gain(j) > 0$, the facility j is opened (if not already open) and reassignments and closures are performed. Let us first define the *distance spare* ds_i as the distance we spare by reassigning the client c_i to the facility f. It is the difference between distances to the current facility and to the facility candidate f. If the difference is negative (current facility lies closer then f) we set $ds_i = 0$. Next we define the *close spare* cs_j as a cost we can spare by closing facility f_j. It is the facility cost minus expenses for reassigning all points from f_i to f. Again if cs_j is negative (we cannot spare anything) we set $cs_j = 0$. The gain function is then computed according to Formula 4.22

$$gain(p) = -fc + \sum_{c_i \in C} ds_i + \sum_{f_j \in F} cs_j \tag{4.22}$$

where fc is the facility cost, or zero if there is a facility already open at f. As stated before, computing function gain takes $\mathcal{O}(N)$ time.

In order to obtain a constant-factor approximation, the described local search technique is repeated $N \log N$ times [12], where N is the number of potential facilities. One local search step takes $\mathcal{O}(N)$ time, so the complete clustering algorithm has an overall time complexity of $\mathcal{O}(N^2 \log N)$. We

believe that the number of iterations could be considerably reduced at the cost of slightly decreased accuracy. This is discussed in Section 7.3.1 starting on page 69.

An algorithm to create the initial solution is also presented in [12]. It is for the general case when the facility cost can be different for each facility. This text assumes uniform facility costs so a different algorithm proposed by Meyerson [103] will be described here. It assumes that all input points are potential facilities, i.e., $C = F$, which is quite common in general clustering problems. Points are taken in random order. A facility is always created at the first one. For every other point, the distance d to the closest already open facility is measured. A new facility is opened at the point with probability d/fc (or one if $d > fc$). Otherwise, the point is assigned to the closest already open facility. The initial solution can be generated in $\mathcal{O}(N^2)$ time.

Figure 4.4 illustrates the process how the initial solution is generated. The image on the left shows the situation after processing the first six points. The image in the middle shows that a new facility has been opened at the seventh point and three more points were processed. It is obvious that the assignment of clients to facilities is not perfect. The image on the right emphasises this by showing a crossing of assignments.

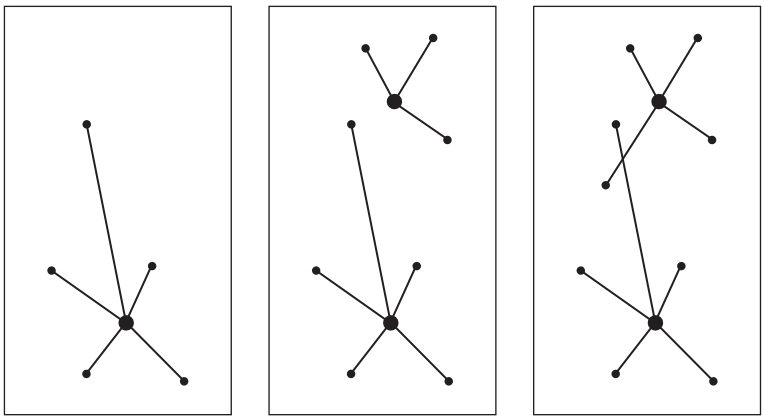


Figure 4.4: The process of generating the initial solution

The complete clustering algorithm can be summarised as follows:

```
Generate the initial solution.

Repeat N log N times
{
  Pick a facility j at random.

  If (gain(j) > 0)
    perform reassignments and closures.
}
```

4.3 Clustering data streams

There are methods for clustering large databases. They mainly focus on reducing the computational time. Although some of them can deal even with data that are larger than the available memory, they are usually not infinitely scalable and therefore are not suitable for true data stream processing. The amount of data in a data stream fairly exceeds the available memory so special techniques are required to process the data in smaller, manageable pieces.

Algorithms for clustering data streams can be divided into three groups. The first approach is based on the divide and conquer strategy. The dataset is divided into several blocks which are solved separately. One or more representatives are then selected from each cluster and these are further clustered to get the result. This technique can be extended to more levels if the dataset is still too large.

The second approach is an incremental clustering. A cluster is created for the first data element. Then following elements are considered one after another. Each one is either assigned to one of the existing clusters or to a new cluster. This is done on the base of some similarity measure. This approach is used by the Leader clustering algorithm [55], for a more recent application see [103]. The major advantage of incremental clustering is that the algorithm does not need to store all the data in the memory. So the space requirements are small. Algorithms are typically non-iterative so their running time is also short. The problem with incremental approaches is that the algorithms are mostly order-dependent. This means that the result of the clustering depends on the order in which the data is presented to the algorithm.

The last approach represents parallel, distributed algorithms. This area is out of scope of this thesis.

This thesis employs the divide and conquer strategy mainly for its hierarchical nature. The following text describes a method proposed by Guha et al. [52] and later addressed in [107, 51]. It is easier to first explain the algorithm that can process short data streams. It partitions the stream into blocks which are processed separately. Each block is clustered by an ordinary clustering algorithm. The method by Guha et al. prefers the local search algorithm for its linear memory requirements. Resulting intermediate cluster centres are given a weight according to the number of elements assigned to them. The algorithm keeps only these centres and discards the remaining data.

If the data stream was longer, the intermediate cluster centres may not fit in the memory. This can be easily solved by allowing multiple passes. The intermediate centres can be stored on an external memory forming another stream, and then processed in the same way as the original data stream. The clustering will be done with respect to the centre weights, i.e., the distance of

a point to its facility is multiplied by the point weight. Weights for resulting centres are then computed as a sum of weights of all assigned intermediate centres. By recursively using the algorithm, it is possible to process a data stream of virtually any size.

It is now straightforward to extend the technique so that no external memory is required and *only a single pass over the data is made.* Pieces of the data stream are loaded into the block B_0 and clustered. The intermediate centres are stored at a higher level in the block B_1. Figure 4.5 illustrates this strategy. The pieces of the data stream are delimited by bold lines. Black dots indicate cluster centres in particular blocks.

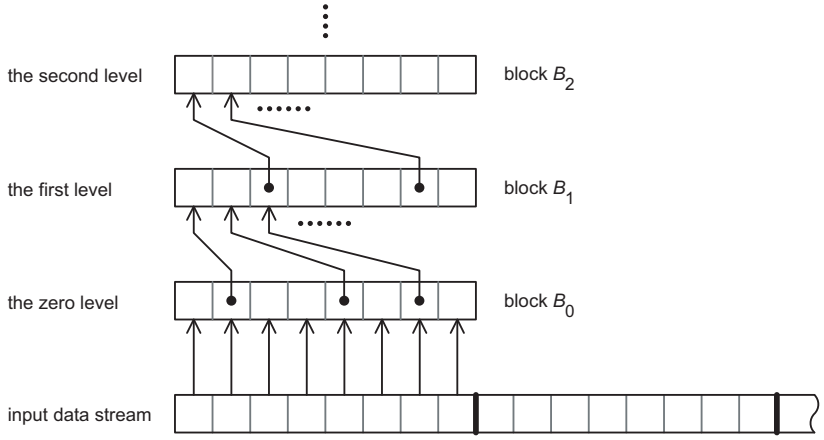


Figure 4.5: Divide and conquer clustering. The data stream is loaded by pieces into the block B_0.

When the block B_1 is full, it is clustered again and the resulting centres are stored at the next level in the block B_2. The algorithm can be written as follows:

1. Read a piece of the input data stream into the block B_0.
2. $process(B_0)$
3. While the data stream is not completely processed, go to 1.

The *process* routine takes a block B_i as a parameter and does the following:

1. Compute the clustering of B_i.
2. Assign weights to the cluster centres.
3. Move the cluster centres to the block B_{i+1} at the higher level.
4. If B_{i+1} is full, $process(B_{i+1})$.

Given a block size m, the algorithm maintains at most m intermediate centres at every level. As soon as there are m centres at any level, they are clustered again and passed one level higher. At the end, the final result is found at the top level after clustering all the intermediate centres.

The number of levels l required to process the entire data stream can be computed as

$$l = \frac{\log(N/m)}{\log(m/k)} \tag{4.23}$$

where N is the number of elements in the whole data stream, m is the block size and k is the average number of clusters in each block. Cited papers present the equation without any further explanation. This paragraph shows how it can be derived. At the zero level, the data stream has N elements which are divided into N/m blocks. These blocks will be clustered into $N/m \cdot k$ intermediate cluster centres, which will be divided into $N/m \cdot k/m$ first level blocks. The situation repeats at higher levels until resulting l level intermediate centres fit into a single block. This can be expressed as

$$N/m \cdot k/m \cdot k/m \cdot \ldots \cdot k/m = 1 \tag{4.24}$$

where k/m repeats l times. After a simple rearrangement

$$N/m = m/k \cdot m/k \cdot \ldots \cdot m/k = (m/k)^l \tag{4.25}$$

Taking a logarithm of Equation 4.25 we get

$$l = \log_{m/k}(N/m) = \frac{\log(N/m)}{\log(m/k)} \tag{4.26}$$

4.3.1 Matching in graph theory

This thesis also presents an approximate algorithm for the minimal Euclidean matching problem. It came from an effort to adapt the clustering for a space partitioning for ray tracing. It turned out that it is better to solve the task as an Euclidean matching. Section 7.5 gives all the details. This section describes the Kuhn-Munkres algorithm [81] which is used for the matching. It is intended as an alternative to the Edmonds optimal algorithm [35] which is rather intricate. The proposed approach builds on the Kuhn-Munkres algorithm for matching in a bipartite graph and uses a Monte Carlo method to adapt the algorithm for general graphs.

A polynomial algorithm for the minimal Euclidean matching is known due to Edmonds [35]. The time complexity is $\mathcal{O}(N^4)$. Gabow [41] proposed an implementation running in $\mathcal{O}(N^3)$ time. The algorithm was designed for arbitrary graphs and is rather complex. The matching method used in this thesis is the Hungarian algorithm developed by Kuhn [81] and adapted by Munkres [104]. The algorithm is known as the Kuhn-Munkres algorithm and

it works for bipartite graphs. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set. The time complexity of the Kuhn-Munkres algorithm is $\mathcal{O}(N^3)$.

The algorithm uses a special vertex labelling to select a subset of edges and create a factor of the graph. A *factor* of a graph is a subgraph with the same vertices and a subset of edges. The algorithm starts with an arbitrary matching. Then it repeatedly attempts to enlarge the matching by swapping suitable edges that belong and do not belong to the matching. If the matching cannot be further enlarged but it is still not a perfect matching, the vertex labelling is adapted, a new factor is constructed and the algorithm starts over. When the matching is a perfect matching in the factor, it is also the minimal weight matching in the original graph.

For the exact formulation of the Kuhn-Munkres algorithm we need a few definitions. Let $G = (X \cup Y, E)$ be a weighted bipartite graph, where X and Y are equally sized sets of vertices and E is a set of edges. We are looking for a matching M from X to Y.

Given the graph G, a *neighbourhood* of vertex set $S \subset X$ is a vertex set $N_G(S) = \{y \in Y | \exists x \in S \text{ so that } \{x, y\} \in E\}$.

Let w_{xy} be the weight of edge $\{x, y\}$. A *feasible vertex labelling* is a function l such that

$$l(x) + l(y) \geq w_{xy} \quad \forall \{x, y\} \in E \tag{4.27}$$

An *equality subgraph* G_l is a factor of G which contains only those edges where $l(x) + l(y) = w_{xy}$.

Let G be a graph, M be a matching in G and P be a path in G. We call P an *M-alternating path* if its edges belong alternatively to the matching and not to the matching. Matching created by *flipping M along P* is a matching created from M by removing all edges in P that belonged to M and adding all edges in P that did not belong to M.

Now, if l is a feasible vertex labelling and M is a perfect matching in G_l, then M is the minimal weight matching in G. See [89] for a proof.

The Kuhn-Munkres algorithm can now be summarised as follows. Start with an arbitrary feasible vertex labelling l and an arbitrary matching M in G_l.

1. If M is a perfect matching in G_l, it is the minimal weight matching in G. Stop. Otherwise, there is some unmatched $x \in X$. Assign $S := \{x\}$ and $T := \emptyset$.

2. If $N_{G_l}(S) = T$, go to 4. Otherwise, choose $y \in N_{G_l}(S) \setminus T$.

3. If y is matched in M, say with $z \in X$, set $S := S \cup \{z\}$ and $T := T \cup \{y\}$, and go to 2. Otherwise, there will be an M-alternating path P from

x to y. Create a larger matching M' by flipping M along P. Set $M := M'$ and go to 1.

4. Compute
$$\alpha_l = \min_{x \in S, y \notin T} \{l(x) + l(y) - w_{xy}\} \tag{4.28}$$
and construct a new feasible labelling l' by
$$l'(v) = \begin{cases} l(v) - \alpha_l & \text{for } v \in S, \\ l(v) + \alpha_l & \text{for } v \in T, \\ l(v) & \text{otherwise.} \end{cases} \tag{4.29}$$
Set $l := l'$ and $G_l := G_{l'}$, choose $y \in N_{G_l}(S) \setminus T$ and go to 3.

Chapter 5

Delaunay triangulation

A triangulation in general is a partitioning of space into simplices – triangles in 2D, tetrahedra in 3D. Let S be a set of points, a triangulation $T(S)$ of the set S has the following properties:

1. All vertices of every simplex are a subset of S.
2. The intersection of any two simplices is either empty, a common vertex, or a common edge, or a common face in 3D.
3. The set of simplices $T(S)$ is maximal, i.e., it is not possible to add another simplex without violating one of the previous conditions.

It follows from the last property that the boundary of a triangulation is the convex hull of S. The Delaunay triangulation is a triangulation where the circumcircle of any simplex is an empty circle, i.e., it does not contain any other point of S. A circumsphere is used in 3D, and a circumscribed hypersphere in higher dimensions.

The Delaunay triangulation [25] is well known in computational geometry, and it is popular for its nice properties. It maximises the minimum inner angle of the triangles in 2D, both locally and globally. In other words, the Delaunay triangulation produces triangles that are closest to the equiangular triangle. This property is important because it ensures a low number of narrow triangles that could cause numerical problems in later processing. In higher dimensions, it minimises the maximal containment sphere of the simplices. In d-dimensional space if no $d+2$ points lie on a common d-sphere and no $k+2$ points, $k < d$, lie in a common k-dimensional subspace, then the Delaunay triangulation is unique.

This chapter recapitulates fundamental properties and methods of construction. Further details may be found, e.g., in [111, 92, 77]. The rest of this chapter focuses on the methods for building hierarchies on the triangulation, and on the methods suitable for triangulation of large data.

5.1 Methods of construction

There are generally six approaches to construct the Delaunay triangulation. The following sections give a brief summary. Alternatively, the Delaunay triangulation can be directly obtained from a Voronoi diagram [136, 111]. This is used very rarely, for instance in [63].

5.1.1 Local improvements

The technique based on *local improvements* [83] first creates an arbitrary triangulation. It then checks the empty circumcircle criterion for all neighbouring triangles, and eventually swaps their common edge if the criterion is violated. Given two neighbouring triangles forming a quad, an *edge swap* is a local modification such that the edge is replaced by the other quad diagonal. In Figure 5.1 on the left, the grey triangle does not fulfil the Delaunay property – its circumcircle contains another vertex. So the bold edge is swapped. Resulting triangles are shown on the right.

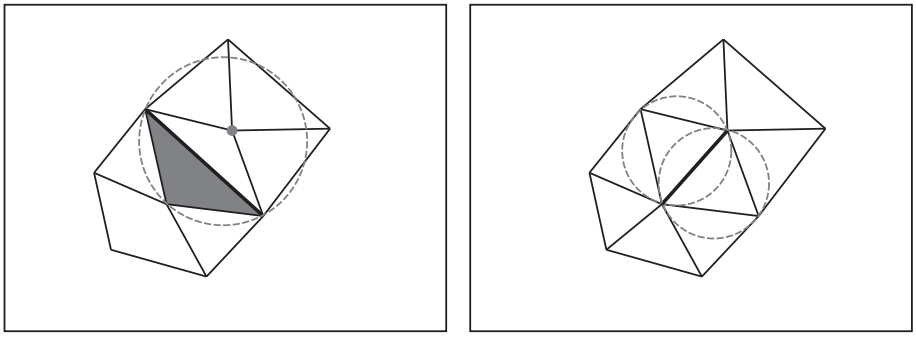


Figure 5.1: Example of an edge swap

The algorithm of local improvements is guaranteed to converge only in 2D. Obviously, it is not suitable for large data processing because the whole triangulation must be held in memory. It also does not allow to have any hierarchy in the triangulation. The time complexity is governed by the number of swaps after the construction of the initial triangulation. In 2D, it is $\mathcal{O}(N^2)$ in the worst case and $\mathcal{O}(N)$ expected.

5.1.2 Incremental construction

Quite a different approach is the *incremental construction*. The fundamental algorithm [101] starts with an arbitrary point of S and its closest neighbour. The edge between these two points forms the base for the triangulation. Then for every outer edge AB of the current triangulation, the algorithm

finds such a point C for which the triangle ABC has an empty circumcircle[1]. The triangle ABC is added to the triangulation. This repeats until the whole set S has been triangulated. The progress of incremental construction is illustrated in Figure 5.2.

The incremental construction has the advantage that the created triangles are never changed. Therefore they can be immediately passed to the output, and the whole triangle mesh does not have to be kept in the main memory. The algorithm has the worst case time complexity of $\mathcal{O}(N^3)$ unless some efficient data structure is used, e.g., a grid as in [16].

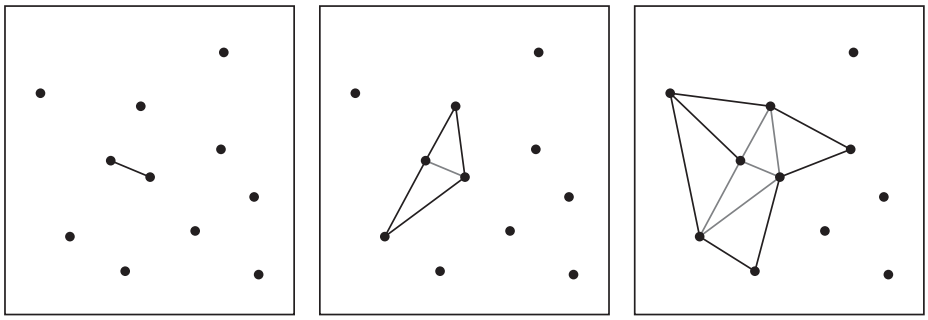


Figure 5.2: The progress of incremental construction

5.1.3 Sweeping construction

A *sweeping algorithm* for the construction of the Delaunay triangulation was presented by Fortune [39]. It is rather complicated but the time complexity is $\mathcal{O}(N \log N)$ in the worst case. An advantage of incremental construction is that once a triangle has been created, it is never changed. This allows to process even large amounts of data. If the algorithm proceeds wisely, the already triangulated parts can be put away, leaving memory for further data. This technique was proposed by Isenburg et al. [70] and is described in Section 5.3.

5.1.4 Incremental insertion

Perhaps the most popular approach is the *incremental insertion*. The algorithm starts with an artificial super-triangle enclosing all the input points. These are then successively inserted into the current triangulation, often in random order.

There are two methods of the insertion. The first one [54, 92] locates the triangle that contains the point being inserted. The triangle is subdivided

[1] Alternatively, a triangle with the smallest circumcircle may be found. It must inherently be empty, because otherwise another triangle with a smaller circumcircle would exist.

into three new triangles formed by two vertices of the original triangle and the inserted vertex. The triangulation is then "legalised" by edge swaps as necessary to satisfy the Delaunay criterion. The insertion with swaps is shown in Figure 5.3. These swaps can affect the whole triangulation in the worst case. The time complexity of the incremental insertion is then $\mathcal{O}(N^2)$. But this happens very rarely, in some special cases designed on purpose to demonstrate this effect. The expected time complexity is $\mathcal{O}(N \log N)$ if a good point location algorithm is used (see Section 5.2).

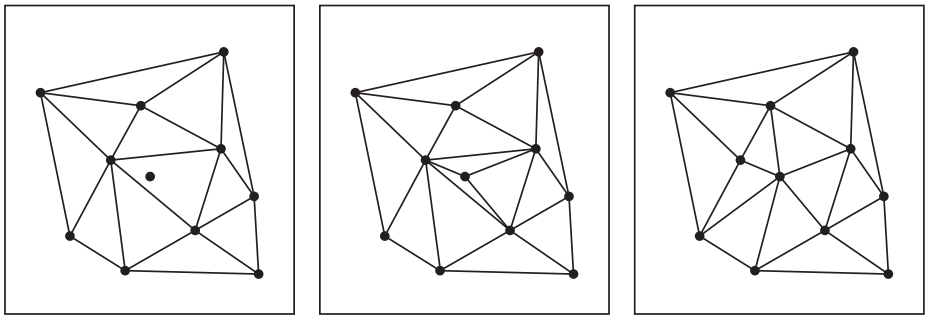


Figure 5.3: Example of point insertion with swaps

Another method of insertion is known as the Bowyer-Watson algorithm [138] with a worst case time complexity of $\mathcal{O}\left(N^{\frac{2d-1}{d}}\right)$. It first finds and deletes all the triangles whose circumcircle contains the point being inserted. The resulting hole is then re-triangulated by creating edges from the hole perimeter to the new point. New triangles fulfil the Delaunay property and no swaps are necessary. The insertion with a re-triangulation is shown in Figure 5.4.

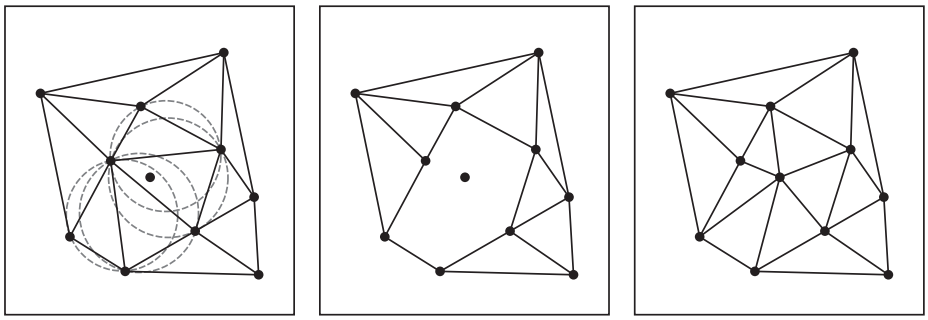


Figure 5.4: Example of point insertion with re-triangulation

The algorithm of incremental insertion is relatively simple and has reasonable numerical stability. When a point is to be inserted, the algorithm needs to locate it inside the triangulation. Several methods exist for this. Section 5.2 describes them in more detail. Incremental insertion does not need to have all the data at the beginning. It is only necessary to know

the bounding box in order to construct the initial super-triangle. All the above properties make incremental insertion well suitable for processing large datasets.

Since the method does not need to know all the points in advance (the points may arrive on the fly), the incremental insertion is well suitable for a dynamic triangulation, i.e., a triangulation where points can be consecutively added and removed. The insertion must be complemented by a method for successive removal of points from the triangulation [138]. This is done by removing all the triangles incident to the point to be removed. The resulting hole is then re-triangulated without using the removed point. The re-triangulation algorithm is commonly called *ear cutting* [27]. An ear is a simplex formed by the vertices of the hole boundary. An ear satisfying the Delaunay property is cut, i.e., the simplex is added to the triangulation, and the hole shrinks. This is repeated until the hole is re-triangulated.

5.1.5 Divide & conquer

The *divide & conquer* strategy is a well known approach and is often used for large data. The most influential works are [53] with the worst case complexity of $\mathcal{O}(N \log N)$, [34] with the expected running time of $\mathcal{O}(N \log \log N)$, and the algorithm DeWall [16] whose complexity was empirically showed to be sub-quadratic. The divide & conquer approach is used for the parallel distributed construction introduced in [78].

Generally, a divide & conquer algorithm recursively divides the data until the pieces are small enough to be easily triangulated, e.g., by the incremental construction. Pieces are processed separately. Results are then merged together to get the complete triangulation. The merging phase is perhaps the most difficult. The pieces must be sewed together by constructing triangles between them. Moreover, some edges already present in particular triangulations may require to be swapped. An example is shown in Figure 5.5.

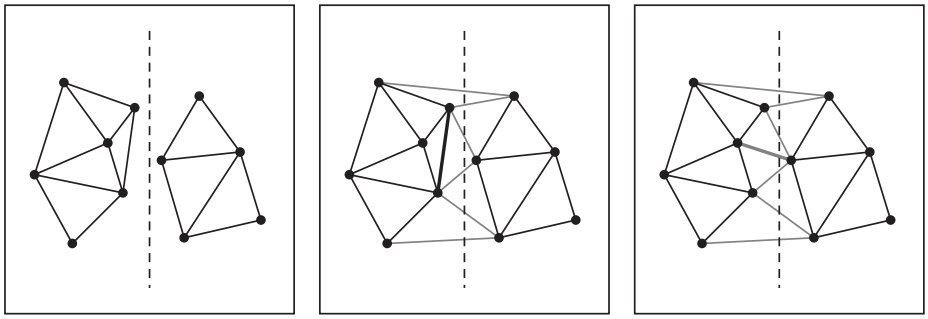


Figure 5.5: The merging phase of the divide and conquer method

5.1.6 Higher dimension embedding

The *higher dimension embedding* is a completely different approach. The input points in dimension E^d are projected on a sphere [8] or on a paraboloid [108] in dimension E^{d+1}. The projection of 2D point P into 3D point P' on the paraboloid can be written as

$$P[x, y] \mapsto P'[x, y, x^2 + y^2] \tag{5.1}$$

A convex hull of the projected points is then constructed. It is proved that projecting the convex hull back to E^d yields the Delaunay triangulation.

The time complexity of higher dimension embedding is determined by the complexity of convex hull construction. The popular gift wrapping algorithm [10] has a complexity of $\mathcal{O}(N^{\lfloor \frac{d+1}{2} \rfloor + 1})$, which is $\mathcal{O}(N^2)$ in 2D and $\mathcal{O}(N^3)$ in 3D. The higher dimension embedding can be used to partition the input data into pieces so that the borders between them are guaranteed to be edges of the Delaunay triangulation. This allows to process large data in a divide and conquer fashion with virtually no merging step [77].

5.2 Point location strategies

It is often necessary to locate a point inside the triangulation, i.e., to find the simplex that contains the given point. This is in particular essential for the incremental insertion algorithm that needs to locate every point being inserted. Traditional point location techniques are the directed acyclic graph (DAG) [92] and the walk. Less common methods include, e.g., the uniform grid [36, 16] or the skip list [143]. The following text describes the walking algorithm which is used in this thesis.

The walk in the triangulation starts at an arbitrary spot in the triangulation, and traverses triangles from neighbour to neighbour until the triangle containing the query point is reached. There are several methods how the walk may proceed [29]. The last point inserted into the triangulation is often selected as the starting point for the next walk. To be consistent with [29], this text denotes the starting point q and the query point p.

The *straight walk* proceeds along the line qp. The algorithm starts with an arbitrary triangle incident to q and turns around q until it reaches the triangle intersected by the line qp. For each following visited triangle, let e be the edge across which the line qp goes out of the triangle. If p lies on the near side of e, the current triangle contains p. Otherwise, the walk proceeds to the neighbour across e. The new vertex of the neighbour is located with respect to the line qp. This determines through which edge of the neighbour the line qp goes out. The straight walk is simple unless it has to deal with degenerate cases. Also numerical stability can be a problem.

The *orthogonal walk* is different in that it first goes along a horizontal line from $q = (q_x, q_y)$ to (p_x, q_y), and than along a vertical line from (p_x, q_y)

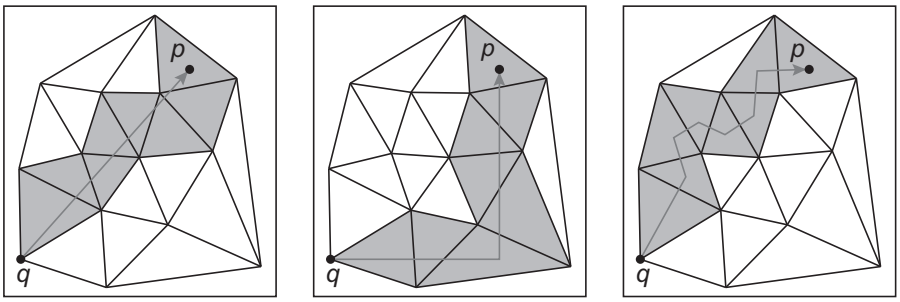
to $p = (p_x, p_y)$. The advantage of this technique is that while walking only in axis aligned directions, it does not need to evaluate expensive orientation tests. Simple greater than/less than comparisons are enough to decide which way to walk. Only at the end of both the horizontal a vertical passes, a few orientation test may be necessary to decide precisely.

The *visibility walk* in its fundamental version starts from any triangle incident to q. Then for each visited triangle it tests the first edge e whether the line supporting e separates the triangle from p. If so, the walk proceeds to the neighbour across e. Otherwise, the next edge is tested. If all three tests fail, the current triangle contains p.

It is necessary to define some edge ordering for this algorithm. But a much more serious problem is that for a non-delaunay triangulation, this walk may fall into an infinite loop. So it cannot be used, e.g., in a constrained triangulation. Fortunately, there is a simple modification – the *stochastic walk* – that overcomes this issue. The only modification is that the edges of a triangle are tested in a random order. This is proved to ensure that the walk will reach p in a finite number of steps.

There is one more improvement called the *remembering stochastic walk*. It remembers the edge it came through to the current triangle. This edge is then excluded from the tests, since it has been already tested in the previous step and it is worthless to do it again. This may spare one orientation test in some triangles.

Figure 5.6 shows an example of the walking strategies – the straight walk, the orthogonal walk, and the stochastic visibility walk. The grey path shows the triangles visited by each particular walk. In this example, all strategies visit the same number of triangles. This is not a rule in a general case.



(a) straight walk (b) orthogonal walk (c) visibility walk

Figure 5.6: Comparison of the walking strategies

The worst case complexity of the straight walk and the orthogonal walk is $\mathcal{O}(N)$ per point location. The stochastic walk can have exponential length. There is an example for every algorithm that demonstrates the worst case [29]. But in practise the expected complexity of all the mentioned walk al-

gorithms is $\mathcal{O}\left(\sqrt[d]{N}\right)$, where d is the dimension. This is nice, although worse than the optimal $\mathcal{O}(\log N)$ complexity of the DAG. The most important is that walking algorithms need a constant (and very little) extra memory, so they are perfectly suitable for processing large data.

Besides the walking, Devillers [28] proposed another approach for point location in a triangulation. It uses a hierarchical data structure. The lowest level holds the complete triangulation T_0. Each higher level contains a triangulation of a small sample of the level below. The walk starts at the highest level k and locates vertex $v_k \in T_k$ that is closest to the query point p. The walk then continues at the levels below (note that also $v_k \in T_{k-1}$) until p is located at the lowest level. This scheme guarantees $\mathcal{O}(\log N)$ location time.

5.3 Streaming Delaunay triangulation

Isenburg et al. [70] introduced a method for computing the Delaunay triangulation of large datasets using only small amount of memory. The method builds on the non-surprising observation that most real-world data has a spatial coherence. If the data is presented in a form of a stream, points lying geometrically close together are also located close together in the stream.

The proposed method exploits this property to introduce *finalisation tags* into the stream. The dataset is divided into regions. When all points from a particular region appeared in the stream, the region is declared finalised and a finalisation tag for that region is injected into the stream. The Delaunay triangulator then uses these tags to identify areas where no more points will arrive and so the triangulation will not change. Such finalised parts of triangulation may be sent to output, freeing memory for further data. The streaming triangulation outputs one big triangular mesh, there is no inherent hierarchy or level of detail.

5.3.1 Inserting finalisation tags into the stream

The finalisation algorithm processes the stream in three passes. In the first pass, it finds the bounding box of the data. A regular grid is then laid over the data partitioning it into rectangular cells. The second pass counts the number of points in each cell. These statistics are then used in the third pass. This time, the algorithm decrements the counters. When a cell's counter reaches zero, all points from the cell have arrived and the finalisation tag for that cell is inserted into the stream.

The spatial coherence can be further increased during the third pass. The algorithm buffers all the points in each cell until the cell's counter reaches zero. All points are then output at the same time followed by the finalisation tag. The increased coherence is well worth the additional memory demands and still requires far less work than fully sorting all the data.

Many algorithms, such as the incremental insertion, may experience a significant drop of performance if the input data arrives in an undesired order. To avoid this it is best to process input points in a random order. So ultimate spatial coherence is not always the best. Therefore the finalisation algorithm samples several points from the stream and promotes them to earlier spots in the stream. This is done locally within each cell and also globally among all cells.

On the local level the algorithm uses the BRIO [2] which was designed exactly for this purpose. In the third pass when a cell is finalised and points released to the output stream, the algorithm moves a sample of randomly selected points to the front of the chunk.

The global sampling starts in the second phase. The algorithm builds a quadtree whose leaves are the grid cells, and stores one point from each quadrant at each level of the tree. During the third pass it moves these sampled points to early spots in the stream. Points are not moved right to the beginning because for very large data this would destroy the spatial coherence. Instead of releasing all the points at once, they are inserted into the stream in a lazy fashion. When a cell is finalised, the sample points of all its ancestors and their immediate children in the quadtree are released before the points of the finalised cell.

5.3.2 Streaming triangulation

The triangulation algorithm uses the common method of incremental insertion. Conventional programs output triangles after processing all the data. The speciality of the streaming algorithm is that it outputs a triangle whenever it determines that the triangle is final, i.e., its circumcircle has no intersection with any unfinalised cell. Such a triangle is for sure in the Delaunay triangulation since no point arriving in the future can be inside its circumcircle. A triangle that is not final is called active.

In addition to the triangulation itself, the algorithm keeps a quadtree that remembers which cells have been finalised. When a point arrives in the stream, it is inserted into the triangulation. When a finalisation tag arrives, the algorithm notes the finalised cell, determines which triangles become final, writes them to the output, and frees their memory. This dramatically reduces the amount of memory used by the program. Figure 5.7 shows the streaming Delaunay triangulation in progress. The points in the white part have been processed and their triangles sent to output. The grey triangles are active. There are a few representative circumcircles that intersect unfinalised cells. The figure shows a situation when points are being inserted into the cell in the middle.

The algorithm uses a straight walk to locate a point inside the triangulation. As can be seen from Figure 5.7, the location may fail because it tries to walk through a final triangle that is no longer in memory. In such a case the

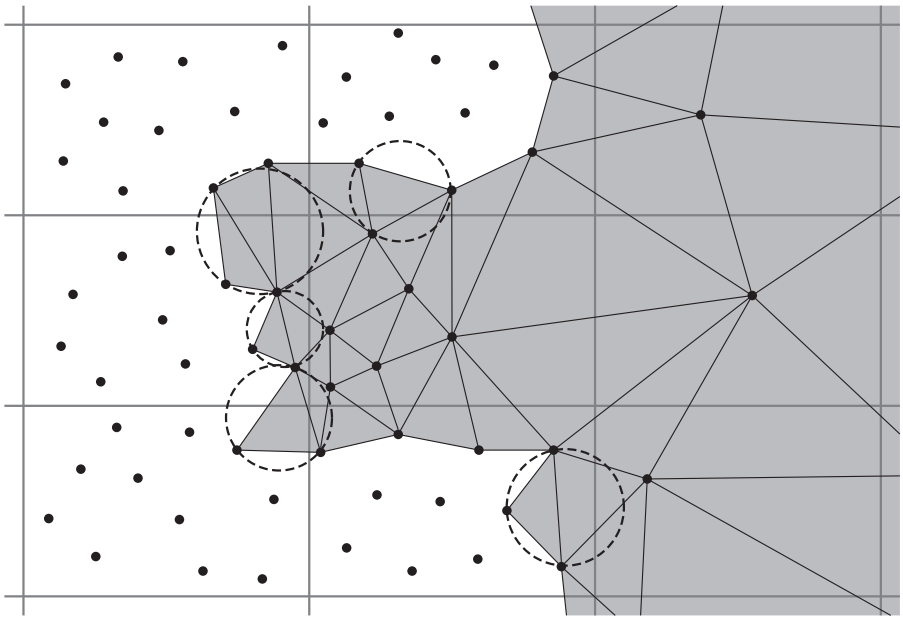


Figure 5.7: The streaming Delaunay triangulation in progress. The image is inspired by [70].

walk is restarted from a different starting point. Each quadtree leaf maintains some triangles whose circumcircles intersect the leaf's quadrant. So the algorithm finds the quadrant containing the point to locate and restarts the walk from one of the triangles on the quadrant's list. If the walk fails again, the algorithm tries other triangles on the list and then triangles from lists of neighbouring quadrants. If no walk succeeds, the algorithm resorts to an exhaustive search through all active triangles. Isenburg et al. claim that this happens for fewer than 0.001% of points.

When the algorithm reads a finalisation tag, it needs to check which active triangles become final. It first checks whether the circumcircle of a triangle is completely inside the cell that was just finalised. This simple test marks many triangles as final. If the test fails, a circle-rectangle intersection is computed. The quadtree hierarchy is used to early reject triangles that are still active. If a triangle's circumcircle intersects an unfinalised cell, it would be wasteful to test the triangle again before that cell is finalised. So the triangle is added into the cell's list and ignored until the cell is finalised. When the proper finalisation tag arrives, the triangle is checked again. Tests continue in the quadtree from the finalised cell where the triangle was listed.

The streaming Delaunay triangulation is very fast. It processes 500 million points in 48 minutes [70]. A drawback is that the input must be approximately sorted, and that the algorithm makes three passes over the data. The result is one huge triangle mesh. It has neither inherent hierarchy, e.g., for a level of detail, nor it is possible to extract just a part of the mesh.

Part II

Contributions

Chapter 6

Summary of the proposed solution

This thesis presents a new approach to handling huge geometric data. The input is a cloud of points. It has neither to be sorted nor to fit into the main memory. Typical input data includes ground-based and airborne laser scanning, the so called LIDAR (Light Detection and Ranging). Except point clouds, the method can be easily adapted to process other data such as a triangle soup. The output is the dataset organised in a hierarchical structure. It offers an efficient access to any part of the data in various resolutions. The hierarchy can be utilised to economically transfer selected parts of the data in the required level of detail. Visualisation can be done using a point-based rendering technique. A multiresolution triangulated model can be built from the hierarchy, and the data can be visualised as a traditional mesh. The triangulation also provides a well known structure for potential further computations.

Point based methods have been developed for rendering large datasets. They are not directly applicable to data manipulation such as triangle mesh construction and editing. There are methods for constructing a triangulation from very large data. The output is one big triangular mesh without inherent hierarchy or level of detail. Many methods exist for simplification and creating the level-of-detail representation. Generally, they require the whole, full resolution triangular mesh at the beginning which is a problem if the data is too large.

The solution proposed in this thesis consists of two stages. The input is preprocessed by a clustering in the first stage. Clusters of points are identified in the data. Each cluster is replaced by a single representative point chosen from the cluster members, typically the centre. Keeping only the representative reduces the amount of data while preserving important features. All the other points are put aside on the hard disk. Unlike simple sampling, the clustering carefully selects which points to drop and which

representative to chose. It has a higher adaptability than a grid or a tree subdivision. The clustering is normally done according to the geometrical distance of the points. More complex criteria are easy to integrate. Namely this thesis offers the elliptical metric described in Section 4.1.1. Further possibilities include colour, curvature, and even non-numeric attributes.

Traditional out-of-core algorithms are designed for large data but they still have limitations and processing really gigantic data is inefficient. Therefore, this thesis employs the data stream approach to cluster large data. Most streaming algorithms dealing with geometry require at least approximately sorted input, and they may require several passes over the data. The method proposed here makes only a single pass over the stream and it does not need sorting. The stream is processed in blocks of say 10 000 points that are easy to deal with. The clustering is done within each block independently. The cluster representatives are accumulated in another block at the so called higher level. When this block is full, the representatives are clustered again and the result is stored at the next higher level.

Data stream clustering algorithms typically keep only the "top level" clusters and discard everything else. The method proposed in this thesis stores and keeps track of all the lower level clusters. One pass over the stream is sufficient to build the hierarchy of clusters. It is stored on the hard disk, so the preprocessing is done only once. Every cluster representative has the address where all the remaining cluster members are stored, so any cluster can be later loaded on demand. The cluster hierarchy constitutes a multiresolution model of the original data. The top level contains the fewest points and represents the coarsest view of the data. The cluster hierarchy can serve as a flexible space partitioning for computations on large data.

A dynamic hierarchical triangulation is constructed in the second stage. This introduces a structure into the data, allowing better visualisation and further computational processing. The triangulation is *dynamic*, i.e., points can be inserted and removed when needed. A *hierarchical* triangulation means that it is built on the hierarchy of clusters. The triangulation of the top level is constructed at the beginning. It gives a coarse model of the whole data. Every vertex represents a cluster of points at a lower level. The cluster can be *expanded*, i.e., all its points are loaded from the hard disk and inserted into the triangulation, which locally increases the level of detail. More clusters can be expanded and the expansions can continue at lower levels down to the original data in the full resolution. Clusters can be later *collapsed*, i.e., the points forming the cluster are removed and replaced by the representative, to free the memory for expansion of other clusters. A new method is presented for removing a number of points from the triangulation at the same time. The expansions and collapses provide an interactive control over the level of detail, the triangulation is updated in real time. The multiresolution model can have a different level of detail in different parts.

The input points may represent a surface of an object, such as laser scanned data, or a volume, such as a reconstruction from computer tomography images. In most cases these objects have a non-convex shape but the 3D triangulation fills the whole convex hull. If only the interior of the object is required, an additional postprocessing is proposed to restore the non-convex shape. It successively removes tetrahedra, starting from the convex hull, proceeding deeper into the model. The removal stops when the probable surface of the object is reached. The remaining tetrahedra constitute the approximated interior of the object. A detailed description is in Section 8.3. This method was designed as simple as possible. It can be easily replaced by a sophisticated surface reconstruction.

Chapter 7

Contributions to clustering

This chapter summarises the contributions I made to the clustering algorithms. Most of them are modifications and improvements to the local search algorithm for the facility location. The implementation of the data stream hierarchical clustering is available as a .NET library [125] under the Creative Commons license.

Experiments were made with clustering pixels in digital images. It was intended for compression or segmentation. Although the method works, it does not make a significant contribution to the state of the art.

7.1 Data stream facility location

The original algorithm by O'Callaghan et al. [107] computes the k-median clustering. The facility location is used to reduce the data at particular levels. A binary search is performed at the end to find such a facility cost that yields exactly k clusters, so the clustering repeats several times. Also an a priori knowledge about the number of clusters in the data is necessary.

One of the modifications proposed in this thesis is to compute the facility location as the result. There is no need for the repeated computation. Dealing with just the facility location seems to be only a part of the original method, but it is not that simple. The clustering is parametrised by the facility cost instead of the number of clusters. A general facility cost value is proposed that gives a reasonable clustering for most datasets used in the experiments. Additionally, the normalisation of point weights is introduced to achieve consistent results throughout all the clustering levels. The following paragraphs describe the facility cost value and the weight normalisation in detail. The proposed method was published in [126].

The original data stream clustering keeps only the "top level" clusters as the result; it discards everything else. This thesis proposes an additional modification to store all the intermediate clusters, building the hierarchy for later use. Details are given in Section 7.2.

7.1.1 Facility cost setting

The facility cost determines how strongly the data will be clustered. A high setting means an aggressive clustering resulting in a low number of large clusters. A low setting will cluster just moderately producing many small clusters. Experiments are usually needed to find the facility cost that best fits particular data. The facility cost is a counterbalance to point distances. A small cost is appropriate for clustering points in a unit square, and a high one for points in $[0; 10^6]$ interval. To avoid time consuming normalization of point coordinates, the facility cost should be derived from the range of point coordinates. The proposed facility cost value is equal to the diagonal of the points bounding box. It mostly produces reasonable results. The cost can be doubled for a stronger clustering or divided by two to get a moderate clustering. The effect of scaling the facility cost is shown in the experiments in Section 7.1.3.

Of course each block of the data stream can have a different bounding box. So the clustering is done in each block with a different facility cost. This is not a problem thanks to weighting the resulting facilities. Clustering with a lower facility cost produces more facilities with a lower weight. A higher facility cost yields fewer facilities with a bigger weight.

7.1.2 Weight normalisation

Another problem comes with clustering at higher levels. As described in Section 4.3, points have weights so all distances are multiplied by some (possibly large) numbers. If we performed the clustering as usual, a facility would be opened at almost every point because their weights make them several times farther from each other. A solution might be to increase the facility cost. But the point weights will grow higher with increasing level which may lead to numerical problems. Therefore the weight normalisation was proposed.

Points at the level zero have unit weight. The goal is to keep the weights around one at the higher levels too. The proposed normalisation simply divides all the weights by their average. Then the average of the normalised weights will be one. It is important to do the normalization of all the points in a block at the same time. That means not earlier then the block is full. Normalizing the weights immediately after clustering a lower level block (before passing the points to the higher level) would be wrong. Each block may have a different number of clusters so the average weight may also vary. Thus points from different blocks would not be normalized equally.

Section 4.3, page 42 describes the *process* routine which takes a block B_i as a parameter. To do the weight normalisation, step 4 of the routine needs to be extended as follows (the new part is typed in boldface):

4. If B_{i+1} is full, **normalise the weights in** B_{i+1} and $process(B_{i+1})$.

7.1.3 Experiments

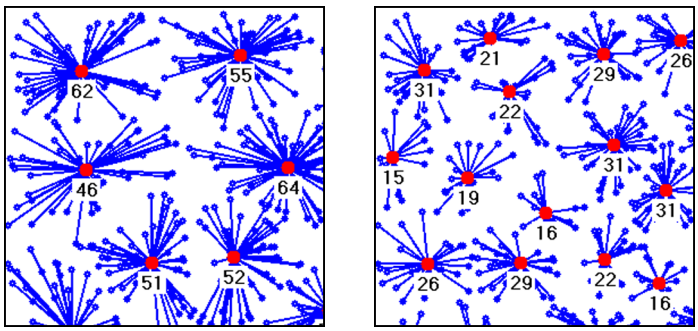
It this section we present experiments made with clustering geometric data. We discuss how the results depend on parameter settings and on the distribution of points in the stream. We give recommendations on how to set the parameters to get a good clustering.

The algorithm was implemented in C# for the .NET Framework 2.0. Experiments were done on Intel Pentium 4 3.2 GHz with 2 GB RAM and a SATA HDD, running Windows XP. All measured times include I/O operations. Memory requirements were measured using the Windows Performance Monitor.

Facility cost setting

A high facility cost means a strong clustering that reduces the amount of data faster. The algorithm may then run with fewer clustering levels, having lower memory requirements and a shorter execution time. Nevertheless, the facility cost is to be determined based on the input data and the desired result, not to optimise the algorithm execution.

Figure 7.1 shows an example of clustering with a facility cost set to double and to half the diagonal of the data bounding box respectively. Numbers show the facility weights (before normalization).



(a) double facility cost (b) half facility cost

Figure 7.1: Clustering with various facility costs. Images cropped.

Input point distribution

Most authors concerned with large geometric data often rely on that input data will be more or less ordered. While this is generally true, the data stream clustering has a significant advantage that it can handle unordered data as well. The results differ slightly for various input orderings but the differences are negligible for the application in this thesis.

If points arrive in order, we could say cluster by cluster, the algorithm processes them successively cluster by cluster. The situation is illustrated in Figure 7.2. It is demonstrated on a small dataset containing 2200 points in four groups of 550 points each. It was processed with facility cost set equal to the bounding box diagonal, and the block size of 750. Frames (a)–(c) show three blocks at level zero. Resulting facilities are passed to a block at the first level shown in Frame (d).

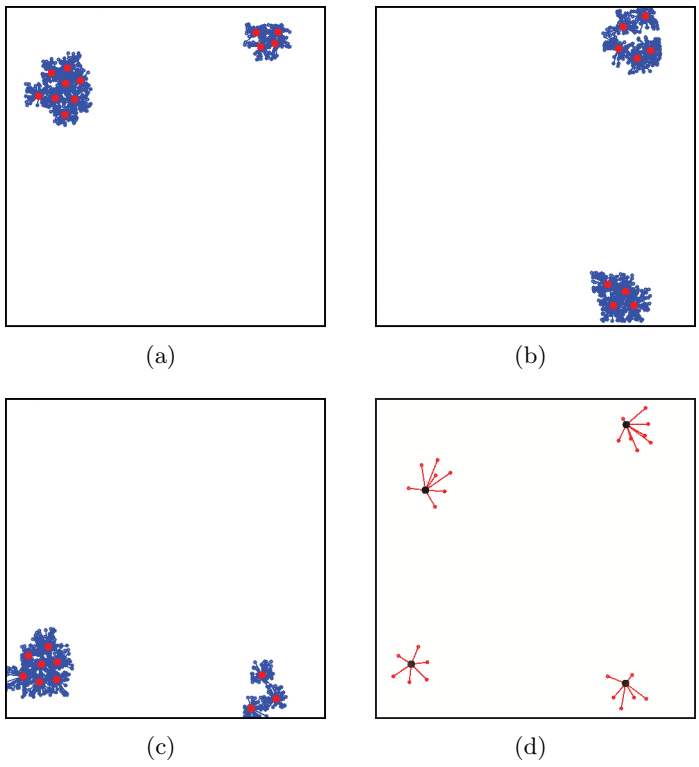


Figure 7.2: Clustering ordered data. Frames (a)–(c) show clusters at level zero. Frame (d) shows the cluster centres from level zero clustered at level one.

Transitions between particular clusters are generally handled correctly. A problem may appear in the rare case when only very few points (call them outliers) of a cluster fall into one block, while all the others fall into another block. The few outliers are not worth opening a new cluster so they will be connected to a different one. An example with a single outlier can be seen in Figure 7.3. Please note this is not an error in the algorithm. It clusters the points exactly according to the given rules. However, for an adversarial input, the result may look unnatural. If this is a problem, it could be solved by a more sophisticated clustering algorithm that is designed to deal with outliers, e.g., [13].

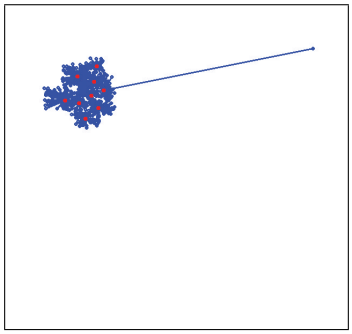


Figure 7.3: Single outlier point connected to a distant cluster

If the input points are scattered in the stream, i.e., they come in a random order, it does not cause any trouble. The algorithm processes several points from different clusters at once. These points form cluster fragments which will be later merged at higher levels. You can see an example in Figure 7.4. It shows the same data as in Figure 7.2, only the order was shuffled. Processing parameters were also the same. Frames (a)–(c) show three blocks at level zero. Resulting facilities are passed to the block at the first level which is shown in Frame (d). The differences from the ordered case in Figure 7.2 are negligible.

To verify algorithm independence of the input ordering, the data was read in a shuffled order rather than sequentially. Random continuous pieces of the stream were read until all the data has been processed. The block size for the data stream processing was 10 000. Two methods of deliberately adversarial shuffling were tried. The first method reads continuous pieces of a random length of 10–10 000 vertices, so it possibly reads just a few consecutive vertices and then it skips ahead. The second method uses a random length of 9 990–10 000, so it reads almost the entire block, but just before the end it skips ahead and reads a few vertices from elsewhere.

Figure 7.5 shows a histogram of point distances to their facilities for the three different scenarios. The left edge of the histogram is zero distance to a facility, the right edge equals to the facility cost (the farthest possible distance). Please note the logarithmic scale on the vertical axis. The histogram shows that a vast majority of the vertices is assigned to very close facilities in all the cases. The first reordering (reading 10–10 000 vertices) gives almost the same result as the sequential processing. The second reordering (reading 9 990–10 000 vertices) has a small influence on the clustering. You can see that some vertices are assigned to more distant facilities. The number of vertices assigned farther than $0.25fc$ is 551 in total which is less than 0.01% of all the data.

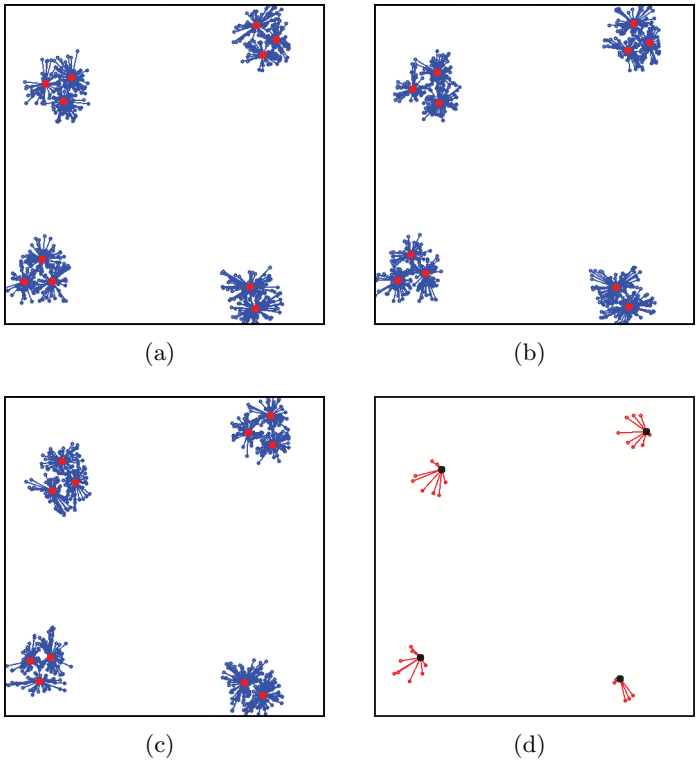


Figure 7.4: Clustering unordered data. Frames (a)–(c) show clusters at level zero. Frame (d) shows the cluster centres from level zero clustered at level one.

Block size

The clustering also depends on the size of block in which the data stream is processed. Block size affects the execution time, the amount of memory used, and also the clustering result. Let N be the number of all points, m be the block size and k be the average number of clusters in each block. The number of all blocks at all levels will be

$$\begin{aligned} & N/m + N/m \cdot k/m + N/m \cdot k/m \cdot k/m + \ldots = \\ &= N/m \cdot [1 + k/m + (k/m)^2 + \ldots] = \\ &= N/m \cdot 1/(1 - k/m) = N/m \cdot m/(m - k) = \\ &= N/(m - k) \end{aligned} \tag{7.1}$$

As proved in [12], $m \log m$ local search iterations are necessary for each block. The total number of iterations over all blocks will be

$$N/(m - k) \cdot m \log m \tag{7.2}$$

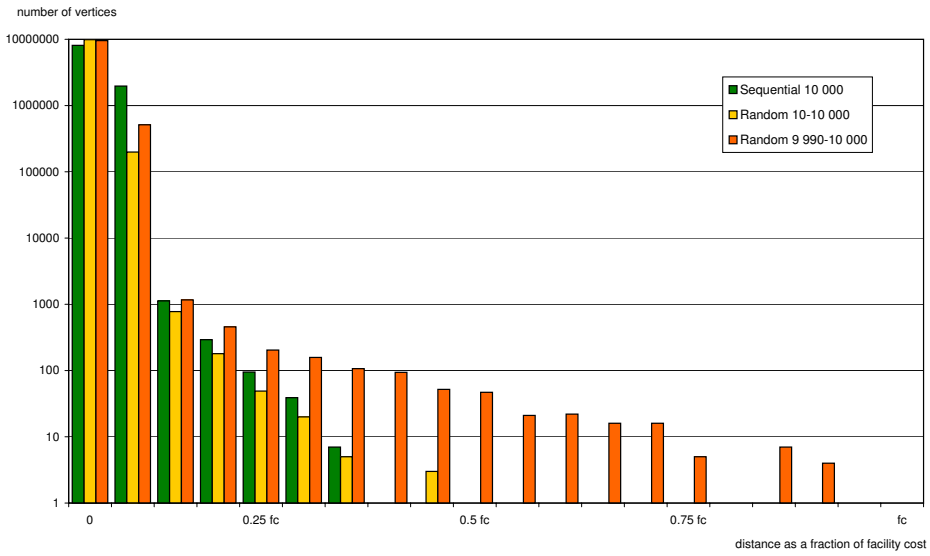


Figure 7.5: Impact of the data ordering on the clustering

As k is proportional to m, we can write

$$N/(m - c_1 \cdot m) \cdot m \log m = c_2 \cdot N \cdot \log m \tag{7.3}$$

where c_1, c_2 are some constants. So by decreasing the block size m, the number of iterations necessary to process the dataset also decreases.

Lowering the block size also decreases the memory requirements. Let l be the number of all the levels needed to process the data stream. The amount of memory required is proportional to

$$m \cdot l = m \cdot \frac{\log(N/m)}{\log(m/k)} \tag{7.4}$$

Since m/k can be considered constant, we can write

$$m \cdot l \approx m \cdot \log(N/m) \tag{7.5}$$

whereas $N \gg m$. The black line in Figure 7.6 shows a graph plot of Equation 7.5 for $N = 10^6$. We must use an integer number of levels (rounded up) in practise. This is shown as the blue line. You can see peaks where the number of levels changes.

It would seem that it is best to process the data in very small blocks, but there is a drawback. The local search algorithm itself computes an approximation of the optimal facility location. Processing the stream in distinct blocks increases the error. The smaller the block, the worse the approximation, see [51] for details. According to the experiments done for this thesis, when varying the block size, the clustering also varies somewhat but the

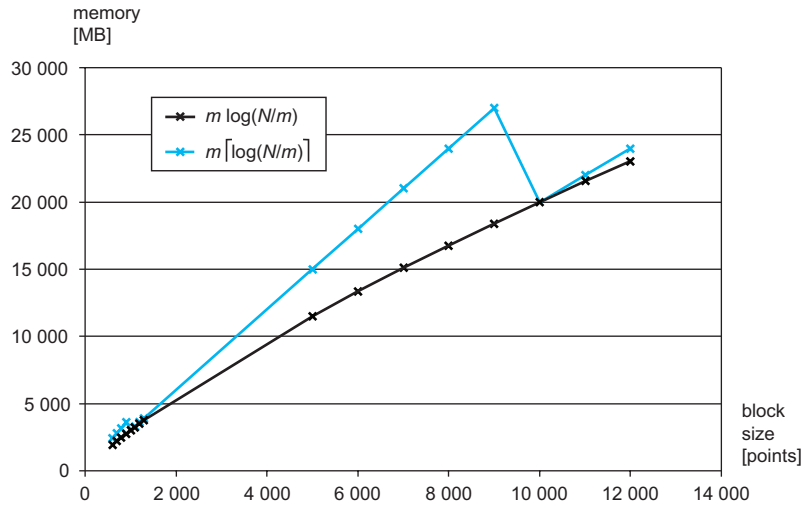


Figure 7.6: Graph plot of memory requirements. Black line is a direct plot of Equation 7.5, blue line is the plot for an integer number of levels.

result still looks natural. The major difference is that clustering in small blocks produces higher number of smaller clusters. Table 7.1 summarizes the experiments with the Lucy model [132] comprising 10 million vertices. The facility cost was set equal to the bounding box diagonal. For the block size of 500, the 516 cluster centres do not fit into the second level so a third level is necessary. With increasing block size, the time needed to process the data increases. The required memory increases too. There is a peak for the block size of 1250 points where the number of levels changes from 3 to 2.

Table 7.1: The influence of block size on the clustering

Block size [points]	Time [h:m]	Memory [MB]	Number of clusters at particular levels		
			level 1	level 2	level 3
500	0:20	2.23	35 126	516	15
1250	1:01	4.44	29 047	404	
2500	1:58	3.97	28 400	366	
5000	3:51	4.09	28 298	337	
7500	5:43	5.27	28 270	336	
10000	7:36	5.78	28 062	303	

7.2 Storing the cluster hierarchy

The original data stream clustering algorithm [52] keeps only the "top level" clusters and discards everything else. In order to make use of the cluster

hierarchy, it is necessary to store all the intermediate clusters; preferably in an easily accessible format. Each clustering level is saved in a separate binary file. This is no problem because even million-sized geometric models can be clustered using only a few levels. So the result will be just several files with a convenient access to particular levels.

Vertices of each cluster are stored in a continuous block as illustrated in Figure 7.7 in the bottom row where you can see clusters distinguished by different shades of grey. Each cluster centre is stored at a higher level along with an address and a size of the block containing the points of the cluster. This is seen in the top row in the figure. Using this structure it is possible to load any particular cluster from an arbitrary level.

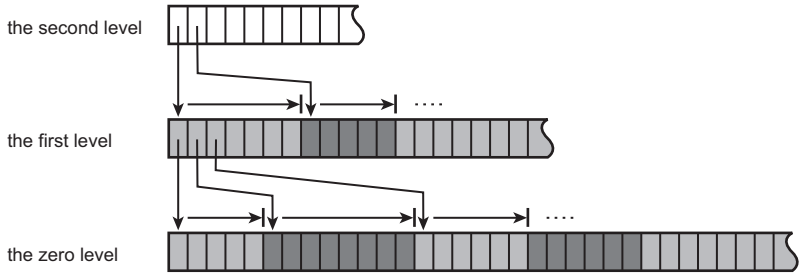


Figure 7.7: Storage scheme of the cluster hierarchy on the hard disk. Cluster centres at the higher levels have pointers to the whole clusters at the lower levels.

Figure 7.8 illustrates the possibilities of the cluster hierarchy. The Lucy model [132] consists of 10 million vertices. The image was rendered using all 1825 vertices from the second level of the hierarchy, 5832 more vertices were loaded from the first level to increase the resolution of both the hands, and another 2501 vertices from the level zero (the full resolution) were added to the index finger tip.

The hierarchy storage scheme was introduced in [124] and later published in [127].

7.3 Speeding up the facility location

The local search algorithm used directly as it is theoretically presented is too slow. This section presents several acceleration techniques to make it more efficient. The algorithm basically consists of three operations. First the generation of the initial solution and then repeatedly computing the gain and eventually performing reassignments. Generating the initial solution is done only once at the beginning and it takes only about 2% of the computing time. The iterative local search algorithm remains, consisting of repetitive gain computation and eventual reassignments. Most of the time, about 96%, is spent evaluating the gain function; see Section 7.3.4. The reassignments take significantly less time and the nature of the operation (reassigning client

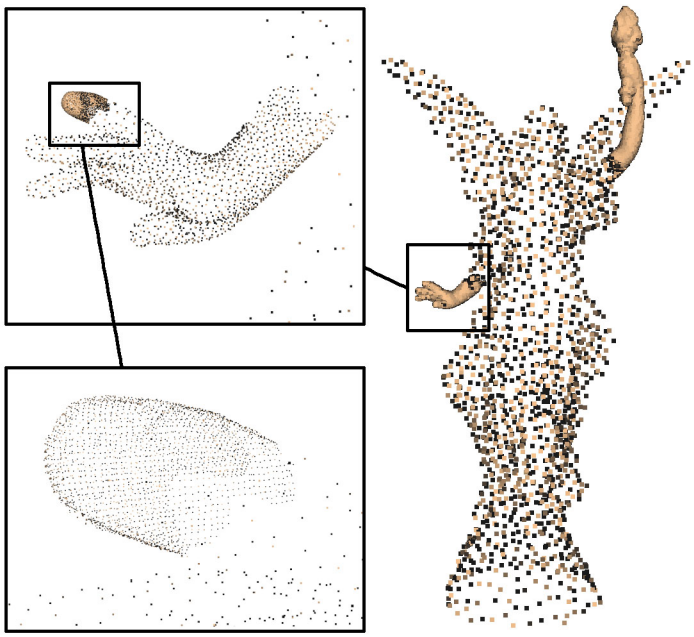


Figure 7.8: Selected vertices of the Lucy model. Data provided by [132].

indices from one array to another) does not give much space for improvements. The effort to speed up the computation was therefore focused on the gain computation. The proposed methods were published in [126, 128].

7.3.1 Reducing the number of iterations

Charikar and Guha [12] proved that $\mathcal{O}(N \log N)$ local search iterations are necessary to achieve a constant factor approximation to the facility location. However, the running time grows unpleasantly if the data stream is processed in large blocks. Experiments with the number of iterations showed that it can be reduced significantly without major impact on the results. Only about $0.1N$ iterations suffice for uniformly distributed data. Even less iterations are required for data which forms obvious clusters.

7.3.2 Space partitioning

To compute the gain of a potential new facility f, clients must be inspected to decide whether f would be closer than their current facilities. It is possible to limit the number of clients that need to be inspected. Given a facility cost fc, any client can be connected to a facility at most fc far away. Otherwise it is cheaper to open a new facility at that client site. Many of the clients are too far from f, so that it is sure their current facility is closer. Eliminating such inperspective clients would be a great benefit. The core idea is therefore

to inspect only those clients that may potentially be closer to the new facility candidate, and thus they can actually contribute to the gain.

Let me introduce the term of *the longest connection*. Let $C' \subseteq C$ be a subset of clients connected to some facilities. The longest connection c_{max} is the maximal distance of any client $i \in C'$ to its facility j. Weighted distance is used for points with weight.

$$c_{max} = \max_{i \in C'} c_{ij} \tag{7.6}$$

The longest connection is the upper bound on the distance where any client from C' can be reassigned without increasing the connection cost.

To derive the upper bound for the whole set C', consider the worst case – a client on the C' boundary. The client can be reassigned at most c_{max} far away from the C' boundary[1]. Therefore, if the distance of the facility candidate f from the C' boundary is greater than c_{max}, then no client from C' can be reassigned to f without increasing the connection cost.

Quadtree, octree, kD-tree

The idea is straightforward – partition the clients using a tree and then inspect only those tree nodes that contain perspective clients. A fundamental space partitioning is a simple quadtree (for 2D) or an octree (for 3D). A kD-tree is perhaps a bit more difficult to build, but it can well adapt to non-uniform distribution of input data. The kD-tree showed particularly good performance on a data in the form of a narrow rectangle.

Either of the trees is built once at the beginning of the clustering. Each tree node stores its bounding box, and the longest connection c_{max} of the clients belonging to the node. Leaf nodes contain in addition a list of clients belonging to them. Figure 7.9 shows an example of a quadtree with the longest connections highlighted in red.

The gain of a facility candidate f is computed by traversing the tree. The distance of f to the bounding box of each node is computed. If the distance is smaller than the longest connection in the node, then the node is traversed. When the traversal gets to a leaf node, all of its clients are inspected.

After considering the relevant clients for reassignment, it is necessary to consider facilities for closure. This cannot be done using the tree because closing a facility spares its facility cost, which can pay for reassigning the clients farther.

Let s be the facility cost *spared* by closing the facility. The clients of the facility are considered one by one. Let c be the distance of a client to its current facility, and let c_f be the distance to the facility candidate. The

[1]The form of the boundary depends on the implementation. It can be, e.g., a bounding box, a bounding sphere, or a convex hull.

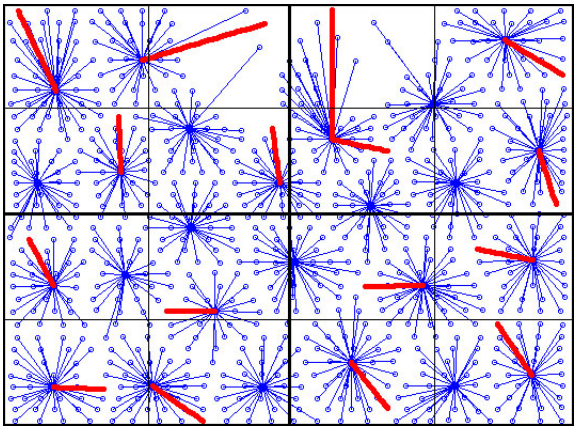


Figure 7.9: Example of a quadtree with the longest connections highlighted

distance extension $c_f - c$ is subtracted from s for each client. As soon as s reaches zero, the facility is not worth closing. This is usually decided after testing just several clients. If s remains positive, it is added to the gain, and the facility is marked for closure.

If the gain of the facility candidate comes out positive, the reassignments and closures are actually performed. This may change the longest connection c_{max} of some tree nodes. If a client with the longest connection is reassigned to a closer facility (c_{max} will decrease), or if any client is reassigned to a farther facility (c_{max} may increase), then the tree leaf is marked that it needs to update c_{max}. Once the reassignments are done, the marked tree leaves are updated. The updates propagate up to their parents.

The tree space partitioning works better for a low facility cost which yields small clusters. The longest connections are short, so only the tree nodes very close to the facility candidate are traversed. Experiments can be found in Section 7.3.4.

Partitioning by facilities

Why to create an artificial space partitioning when there is one already constructed? The clustering itself – although not finished yet – is a partitioning, and it perfectly corresponds to the task being solved.

Each facility (cluster centre) keeps the list of its clients, and the longest connection c_{max}. The cluster boundary is a sphere centred at the facility with the radius c_{max}. To compute the gain of the facility candidate f, clusters are considered one by one. It is to be decided whether f is at most c_{max} from the cluster boundary, that is at most $2c_{max}$ from the facility. See Figure 7.10 for an illustration. If f lies close enough, all the clients of the facility are inspected. Otherwise, the facility is only considered for closure. The algorithm is the same as described for the trees.

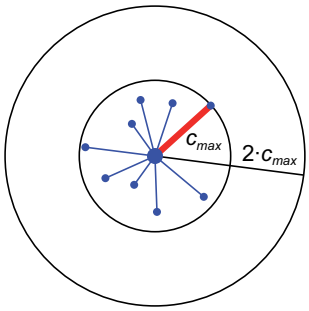


Figure 7.10: Facility with its boundary and the longest connection

If the gain of the facility candidate comes out positive, the reassignments and closures are performed. As in the case of the trees, if a reassignment changed the longest connection of a facility, the facility is marked. Once the reassignments are done, the marked facilities are updated.

The cluster size also matters for the partitioning by facilities. A great facility cost yields large clusters. The longest connections are long, so even the clusters far away from the facility candidate must be inspected. A low facility cost and small clusters is also not good. There is a lot of clusters, and, although most of them are too far from the candidate, the overhead grows. Experiments can be found in Section 7.3.4.

7.3.3 Parallelisation

Modern computers commonly have four or more CPU cores. This gives the possibility to employ parallelism to further accelerate the computation. This section describes two possible approaches to the parallelisation. Both of them suppose the use of the space partitioning by facilities (see Section 7.3.2) because the parallelisation is straightforward.

Single gain computed in parallel

Thanks to the space partitioning, the gain is computed facility by facility. The computation for each facility is independent. Therefore, the most straightforward parallelisation is to divide the set of facilities among several threads and let every thread deal with a subset of facilities. The computation can proceed independently, so no synchronisation is necessary. When all the threads finish, their particular results are easily merged – the gain values are summed up, the lists of clients to reassign are concatenated, and the lists of facilities to close are concatenated.

Parallel computations of several gains

The gain often gives a positive result in early iterations of the algorithm. With increasing number of iterations performed, positive results become less frequent. In later stages of the algorithm, gain results come out mostly negative. Again, the performance is affected by the cluster size. A detailed evaluation can be found in Section 7.3.4.

This gives another possibility of parallelisation – to compute the gain for several different facility candidates simultaneously. Candidates with a negative gain are useless. No reassignments are done, so the clustering is not modified. Therefore, any number of negative results can be accepted as valid iterations of the local search algorithm at the same time (yet unsuccessful to improve the solution). This means that actually several iterations of the clustering algorithm are executed in parallel. The computation pauses only when a positive result appears, and the clustering is modified.

If the gain of any of the candidates is positive, the appropriate reassignments are done. Other candidates with a negative gain can be accepted as well. But if more candidates happen to have a positive gain, the one with the greatest gain is used. The remaining positive gain results must be discarded because the clustering changes after the reassignments to the first candidate, so the other positive results are not valid anymore.

7.3.4 Experiments

This section summarises the experiments done to identify the bottlenecks, to find improvement possibilities, and to document the achieved speedup. The program is written in C# under the .NET Framework 2.0. The experiments were done on several computers with a different hardware configuration. Please see the respective subsections for details.

Three types of datasets were used for the experiments. The first type are digital elevation maps (digital terrain models). The largest one is the Earth (309.4 million points) [139]. The Grand Canyon (8.4 million points) [100] and the Puget Sound (268.5 million points) [133, 88] are fragments extracted from a high resolution map; they are available from the Georgia Tech [43]. The second type are surface models from laser scanning. Namely Lucy (10.1 million vertices), David (28.2 million vertices), St. Matthew (187 million vertices), and the new (2011) scan of David with a quarter millimetre resolution (468.6 million vertices); all from the Digital Michelangelo project [132]. The third type is a full 3D model (not only surface) of the Power Plant (11.1 million vertices) [137].

Performance on large data

The amount of memory required to process a data stream can be estimated as follows. The number of levels l is computed using Equation 4.23 presented

in Section 4.3 on page 43. The block of points and the opened facilities need to be maintained at each level. There are m points and k facilities. A point takes 56 bytes in my implementation. A facility takes 24 bytes plus a list of points assigned (4 bytes per point). Every facility generally has a different number of points, but every point must be assigned somewhere, so the total number of point references is always m, ergo $4m$ bytes. The amount memory Mem is then computed as

$$Mem = l \cdot (56m + 24k + 4m) \tag{7.7}$$

Take the Earth dataset [139] for example. It has $N = 310$ million points; let us say it will be processed in blocks of $m = 5\ 000$ points, and the number of clusters will typically be 1% of the input points, thus $k = 0.01m = 50$. From Equation 4.23, the number of levels $l = 3$. So the memory used by the algorithm is

$$Mem = 3 \cdot (56 \cdot 5\ 000 + 24 \cdot 50 + 4 \cdot 5\ 000) = 903\ 600 \text{ bytes} \tag{7.8}$$

Figure 7.11 shows the clustering of the digital elevation map of the Earth.

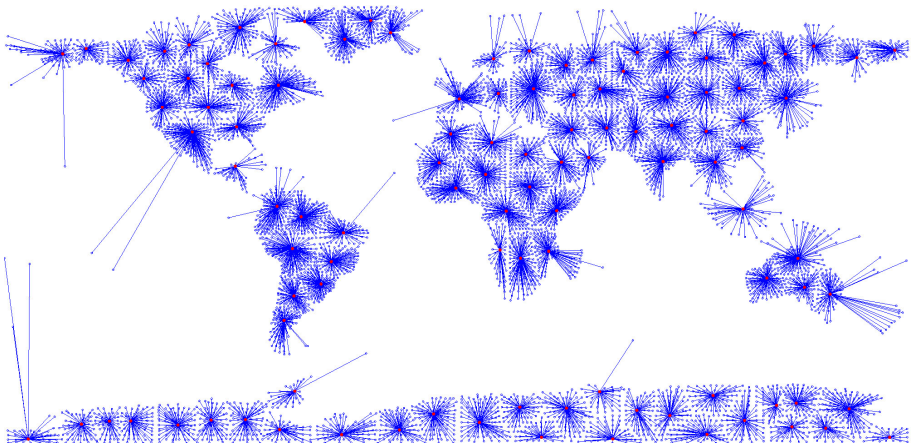


Figure 7.11: Clustering of the whole world. Data provided by [139].

The experiments were performed on Intel Core 2 Quad 2.4 GHz with 4 GB RAM and a SATA HDD, running Windows XP. The clustering was done by blocks of 10 000 points, the facility cost was set to double the diagonal of the bounding box.

Out-of-core processing of large data inherently needs a lot of I/O operations, i.e., reading and decoding the input, and writing the results. Table 7.2 shows the total running time compared to the time spent only by computing the clustering. The I/O operations take up to 10% of the total running time. The percentage varies depending on the format of the input data.

Table 7.3 shows the amount memory used by the data stream clustering. The memory use was measured using the Windows Performance Monitor.

Table 7.2: Comparison of running times with and without I/O operations

Dataset	Number of vertices	Time with I/O [h:m:s]	Time without I/O [h:m:s]	Share of I/O
Grand Canyon	8 M	26:55	25:11	6%
Puget Sound	268 M	14:15:24	13:22:55	6%
Earth	309 M	16:09:24	15:16:07	6%
Lucy	10 M	33:03	30:36	7%
Power Plant	11 M	36:22	33:13	9%
David	28 M	1:31:29	1:24:12	8%
St. Matthew	187 M	10:01:11	9:14:39	8%
David (2011)	469 M	25:52:53	23:12:01	10%

The table also shows the number of vertices at particular levels to demonstrate the hierarchical reduction of the data.

Table 7.3: Time and memory used by the data stream clustering

	Number of vertices				Memory
Dataset	input	level 1	level 2	level 3	[MB]
Grand Canyon	8 M	37 991	318		89
Puget Sound	268 M	1 173 166	10 054	144	163
Earth	309 M	1 065 485	8 085		87
Lucy	10 M	64 978	802		81
Power Plant	11 M	52 084	350		83
David	28 M	115 033	867		85
St. Matthew	187 M	689 944	6 610		84
David (2011)	469 M	1 861 811	17 140	187	97

The real memory use is significantly higher than the estimate given by Equation 7.7. This is due to the user interface and the .NET Framework where the program is running. The program takes about 14 MB of memory upon startup. The Puget Sound and David (2011) must have been processed at level 3 so they required more memory. The Grand Canyon and the Puget Sound have even higher memory requirements because the datasets are stored as PNG images and the image reader loads large tiles into the memory.

Facility location running time

Smaller sized input data was used to identify the bottlenecks in the algorithm. The experiments were done on Pentium 4 3.2 GHz with 2 GB RAM, running Windows XP. The the following datasets were used: Bunny and

Armadillo [132] (laser scanned 3D models), the famous Utah teapot[2] (vertices sampled from the surface), and the Crater Lake [22] (nearly 2D terrain model).

Table 7.4 shows the running time of the original, unoptimised algorithm, and partial times for the relevant parts of the computation. Evaluating the gain functions takes approximately 96% of time. Generating the initial solution and performing the reassignments are negligible.

Table 7.4: Total running time of the original algorithm, and partial times for the important parts.

Dataset	Number of vertices	Total time [s]	Initial solution [s]	Gains [s]	Reassignments [s]
Bunny	35947	23.1	0.7	21.5	0.1
Teapot	80203	113.5	2.3	107.7	0.5
Crater	100001	175.1	2.8	167.9	0.5
Armadillo	172974	524.3	9.0	515.1	1.4

Reducing the number of iterations

It is to be noted that the idea of reducing the number of iterations was studied much earlier than the space partitioning and parallelisation techniques described in the following Sections 7.3.2 and 7.3.3. The speedup discussed here was achieved solely by reducing the number of iterations.

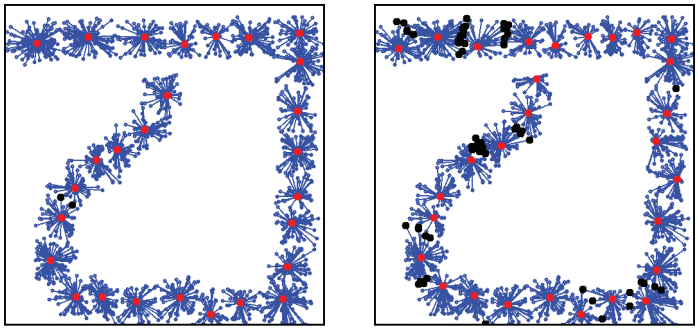
Figure 7.12 shows two examples of clustering. The small dataset for demonstration contains 1640 points. They were processed in a single block with the facility cost equal to the bounding box diagonal. Black dots mark the wrongly assigned points, i.e., points that are connected to a different facility than to the closest one. It is an easily comprehensible visualisation of the error. Please note that it takes into account only the current set of open facilities. No black dots mean optimal assignment to the *currently open* facilities. The clustering may be even better with a different set of open facilities.

Figure 7.12 a) shows the result after $N\lceil \log N \rceil = 6560$ iterations, the overall clustering cost is 187. Figure 7.12 b) shows the result after $0.1N = 164$ iterations, the clustering cost is 194.6 which is 4% more then a).

Table 7.5 summarizes the experiments with the Lucy model [132]. The tests were done on Pentium 4 3.2 GHz with 2 GB RAM, running Windows XP. The facility cost was set equal to the bounding box diagonal. The table lists the time required for $N \log N$ and $0.1N$ iterations. The clustering cost is slightly higher in each block when the reduced number of iterations is done. The table records statistics about this error to quantify the impact

[2] Original 3D model by M. Newell, 1975.

on the clustering quality. The average error stays below 2% in all cases. The maximal error is up to 5% for block sizes of 2500 points and more, so the reduced number of iterations is well usable. The maximal error raises significantly for smaller blocks, so the full number of iterations is more appropriate there.



(a) (b)

Figure 7.12: Clustering results: (a) after 6560 iterations, (b) after 164 iterations

Table 7.5: Experiments on the number of iterations. Times were measured for the original non-accelerated algorithm.

Block size [points]	Time for $m \log m$ [h:m:s]	Time for $0.1m$ [h:m:s]	Percent of time used for $0.1m$	Min. error	Avg. error	Max. error
500	0:20:18	0:01:33	7.6%	0	1.3%	29.9%
1250	1:01:24	0:02:41	4.4%	0	1.5%	14.8%
2500	1:58:00	0:04:29	3.8%	0.2%	1.6%	5.0%
5000	3:51:01	0:07:45	3.4%	0.5%	1.7%	4.9%
7500	5:43:21	0:11:09	3.3%	0.5%	1.7%	3.5%
10000	7:36:36	0:14:32	3.2%	0.8%	1.6%	3.2%

Space partitioning and parallelisation

This section documents the speedup achieved by implementing the proposed improvements. The reduced number of iterations (see Section 7.3.1) was used in all the cases. The experiments were performed on Intel Core 2 Quad 2.4 GHz with 4 GB RAM and a SATA HDD, running Windows XP. The parallel computations were executed in four threads. The clustering was done by blocks of 10 000 points, the facility cost was set to double the diagonal of the bounding box. The time measurements do not include I/O operations so as to compare only the algorithm execution times.

Table 7.6 shows the speedup by space partitioning. The kD-tree is the best overall because it splits the data adaptively. The space partitioning by facilities is a small bit slower, but it is easier to implement and the following parallelisation is straightforward. Table 7.7 shows the speedup achieved by the parallelisation compared to the (non-parallelised) partitioning by facilities. Although ran on a 4 core CPU, the program is not 4 times faster because only the gain evaluation runs in parallel. Eventual reassignments remain sequential. The CPU utilisation oscillated between 25% and 95%. The table shows that the parallel computation of multiple gains is definitely faster. The implementation is also a bit easier.

The influence of cluster size

This section shows how the cluster size affects the efficiency of the proposed algorithms. It is to be noted that the cluster size (the facility cost) is a user specified parameter, and it is therefore unreasonable to search for an optimum.

The graph in Figure 7.13 shows how often the gain comes out positive for various facility cost values. This strongly influences the efficiency of the parallel computation of several gains. At the beginning, gains are all positive because the initial clustering is very coarse, and almost any facility candidate can improve it. Later, the ratio of positive gains drops, especially for great facility costs (large clusters).

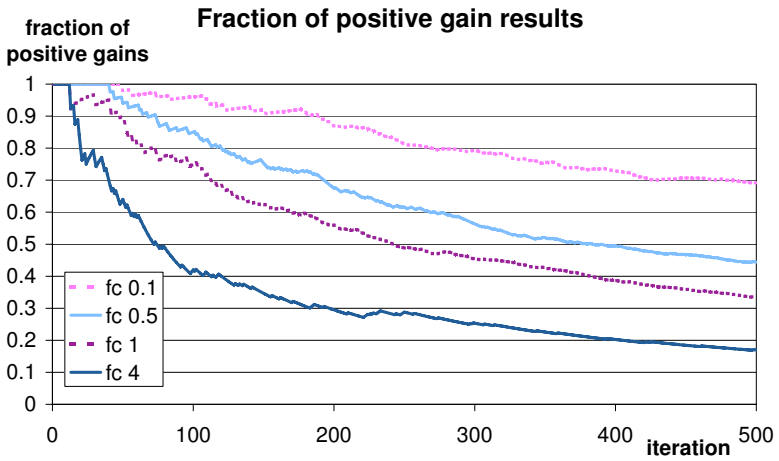


Figure 7.13: Fraction of positive gain results

The graph in Figure 7.14 shows the fraction of *vertices* inspected for various facility cost values. A *vertex* here means either a facility or a client. In both cases, inspecting means computing the distance to the facility candidate. If a facility is close enough, all its clients are inspected. The notion of *vertices* was introduced to overcome the issue described at the end of Sec-

Table 7.6: Speedup achieved by various space partitioning

Dataset	Number of vertices	Original time [h:m:s]	Quadtree time [h:m:s]	Quadtree speedup	Octree time [h:m:s]	Octree speedup	kD-tree time [h:m:s]	kD-tree speedup	Facility time [h:m:s]	Facility speedup
Grand Canyon	8 M	25:11	16:46	33%	17:28	31%	5:35	78%	6:18	75%
Puget Sound	268 M	13:22:55	8:57:57	33%	9:18:06	31%	2:50:41	79%	3:11:13	76%
Earth	309 M	15:16:07	9:56:24	35%	10:19:27	32%	2:40:53	82%	2:34:29	83%
Lucy	10 M	30:36	20:48	32%	21:23	30%	7:01	77%	7:30	76%
Power Plant	11 M	33:13	22:11	33%	22:43	32%	6:54	79%	7:20	78%
David	28 M	1:24:12	56:10	33%	56:37	33%	15:41	81%	15:23	82%
St. Matthew	187 M	9:14:39	5:58:52	35%	6:12:31	33%	1:37:10	83%	1:29:00	84%
David (2011)	469 M	23:12:01	15:00:38	35%	15:33:26	33%	4:01:07	83%	3:41:52	84%

Table 7.7: Speedup achieved by parallelising the gain computation by facilities

Dataset	Vertices	Facility time [h:m:s]	Parallel single gain time [h:m:s]	Parallel single gain speedup	Parallel multiple gains time [h:m:s]	Parallel multiple gains speedup
Grand Canyon	8 M	6:18	4:04	35%	3:06	51%
Puget Sound	268 M	3:11:13	2:01:31	37%	1:32:26	52%
Earth	309 M	2:34:29	1:56:06	25%	1:20:43	48%
Lucy	10 M	7:30	4:43	37%	4:03	46%
Power Plant	11 M	7:20	4:51	34%	3:57	46%
David	28 M	15:23	10:51	30%	8:07	47%
St. Matthew	187 M	1:29:00	1:05:28	26%	46:07	48%
David (2011)	469 M	3:41:52	2:42:07	27%	1:58:15	47%

tion 7.3.2. The *vertices* represent all the facilities and clients we effectively have to deal with. The ratio of inspected vertices is greater at the beginning because of the coarse initial clustering. The ratio is lower for a small facility cost (small clusters).

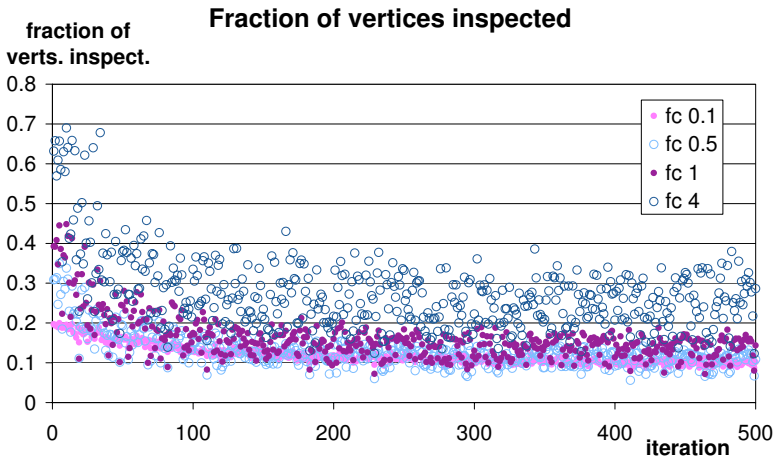


Figure 7.14: Fraction of vertices inspected

7.4 Elliptical metrics

As discussed in Section 4.1, elliptical metrics are an effective feature in both clustering and triangulation. An elliptical metric can be viewed as an elliptical elongation of the classical Euclidean space. Concerning clustering, it allows the clusters to have an ellipsoidal shape rather than spherical. Ellipsoids can significantly better match phenomena like those mentioned in Section 4.1. The effect on a Delaunay triangulation is that edges tend to go along the direction specified by the ellipse. This could be useful to prepare the triangulation for later deformations so as to avoid degenerate cases.

A small contribution of this thesis is the proper derivation of the required mathematical equations. It is presented in Sections 4.1.2 and 4.1.3 along with the description of the original method [134] to keep the mathematics together.

Different elliptical metrics can be defined in limited regions of the data. This adds even more flexibility to the solution. Figure 7.15 shows an example. The first frame shows the data with three regions defining three different elliptical metrics as illustrated by the ellipses; standard Euclidean distance is used outside the regions. The next two frames show the clusters and the triangulation created using the specified metrics.

Figure 7.16 shows an example of using three different elliptical metrics on the Crater Lake [22]. You can see how the cluster shape as well as

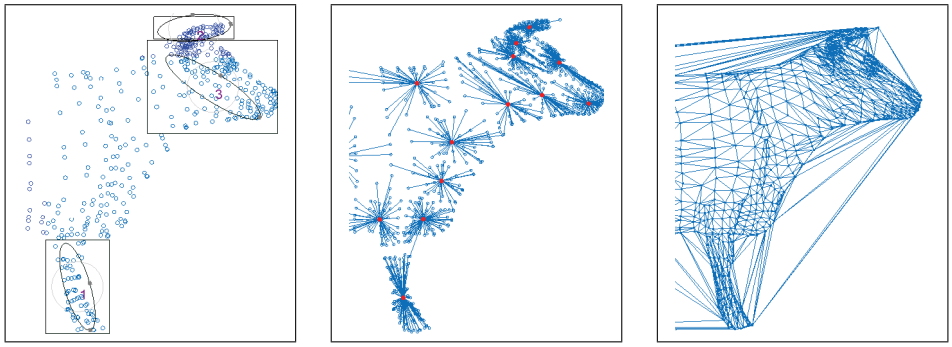


Figure 7.15: Illustration of regions with different elliptical metrics (left). The clustering and the triangulation computed using the defined metrics (middle and right).

the triangulation edges adapt to the crater perimeter. The boxes show the regions where a particular metric is defined; standard Euclidean distance is used everywhere else. The black ellipses define the metric; the grey circles are unit circles for reference.

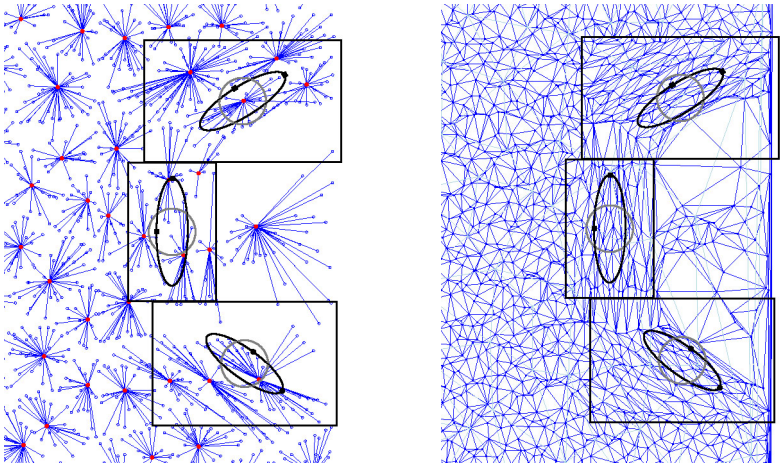


Figure 7.16: Clustering and triangulation computed using three different elliptical metrics in three different regions

The elliptical metrics were published as an extension for the clustering and the Delaunay triangulation in [127].

7.5 Euclidean matching

The proposed clustering methods for manipulating large data made quite a good impression and several people were interested in it. One of the inspiring ideas came from Jiří Bittner, CTU Prague, to use the clustering as a

space partitioning for ray tracing acceleration. However, the partitioning is required to be binary such as in the kD-tree which is often used in ray tracing. The binary partitioning can be translated as: each cluster should have exactly two members. It turned out that adapting the clustering algorithm to such a demand would require substantial changes. I decided to develop another method.

The task is: Given a set of vertices, group all of them into pairs so that the sum of distances between the paired vertices is minimal. An example of such pairing is shown in Figure 7.17. After a brief research the problem

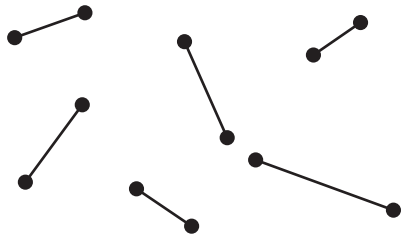


Figure 7.17: Example of minimal Euclidean matching

was identified as the minimum Euclidean matching. It is a problem of graph theory, formally stated as follows. Given a set of $2N$ vertices corresponding to nodes of a complete graph with edge weights equal to Euclidean distances between the respective vertices, find the minimum weight perfect matching. A *perfect matching* is a matching where every vertex is incident to exactly one edge of the matching.

The following was a joined work with student Jan Hyka who participated on the algorithm design and did most of the programming.

Edmonds introduced an algorithm [35] that finds an optimal solution in $\mathcal{O}(N^4)$ time. Gabow [41] proposed a more efficient implementation running in $\mathcal{O}(N^3)$ time. However, the algorithms turned out to be too intricate to implement just as a trial. We chose to implement a heuristics using a matching in a bipartite graph. A *bipartite graph* is a graph whose vertices can be decomposed into two disjoint sets U and V such that no two vertices from the same set are connected by an edge. See Figure 7.18 for an example.

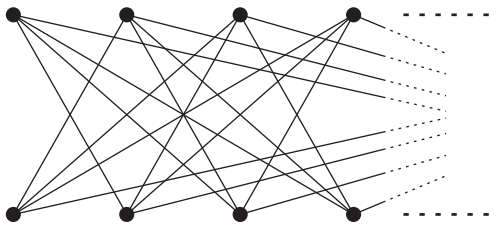


Figure 7.18: Bipartite graph where the perfect matching will be constructed

7.5.1 Randomised partitioning & matching

The minimal weight matching in a bipartite graph is relatively easy to solve by the Kuhn-Munkres algorithm [81, 104] in $\mathcal{O}(N^3)$ time; see Section 4.3.1 for details. The problem is that we generally do not have a bipartite graph so we need to forge one. A Monte Carlo method [102] was developed. It is a stochastic technique for solving problems for which an analytical solution is unknown or is too complex. In general, a Monte Carlo method performs statistical simulations using random numbers. To be more specific, in our case it repeatedly generates random possible solutions and evaluates their cost. At the end, the best solution is selected. The proposed method is reliable and gives satisfactory results.

Each iteration of the Monte Carlo method starts by randomly distributing all the vertices into two equally sized sets. Each vertex in one set is connected to all the vertices in the other set. The matching is then constructed by the Kuhn-Munkres algorithm. This process is run repeatedly. According to empirical experiments, see Section 7.5.2, it is necessary to perform N or perhaps $10N$ iterations. At the end, the solution with the lowest sum of pairwise distances is accepted. Figure 7.19 shows several instances of the matching with the best one framed in bold. The number at lower right shows the sum of distances.

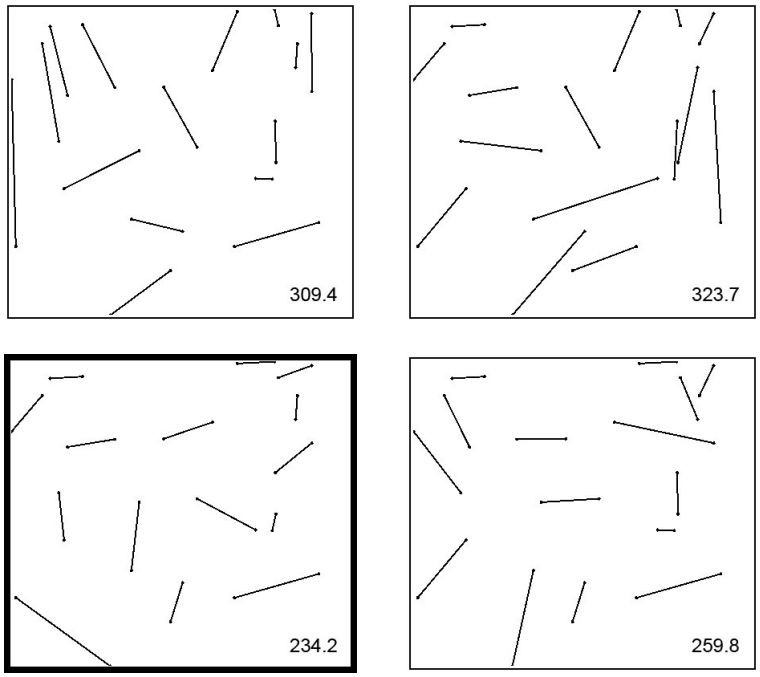


Figure 7.19: Example of several matching instances tried by the Monte Carlo method

The algorithm can be formally stated as follows. We are looking for the minimal Euclidean matching $M_{\min}$.

1. Initialise the minimal cost $c_{\min}$ to positive infinity.
2. Randomly distribute the vertices into two equally sized sets X and Y. Make the set of edges $E = \{\{x, y\} | x \in X, y \in Y\}$, i.e., make an edge from every vertex $x \in X$ to every vertex $y \in Y$.
3. In the bipartite graph $G = (X \cup Y, E)$ construct matching M by the Kuhn-Munkres algorithm. Let the cost of the matching be c.
4. If $c < c_{\min}$, set $c_{\min} := c$ and $M_{\min} := M$.
5. While a stopping condition has not been met, go to 2.
6. Output $M_{\min}$.

Figure 7.20 shows an example of the matching including the built hierarchy. The levels of the hierarchy are coloured from red (the lowest) to black (top) for a better readability. The first frame shows the matching alone, the second one shows the progress of building the hierarchy, finally the last frame shows a complete tree. Especially the middle frame demonstrates that the space partitioning works well.

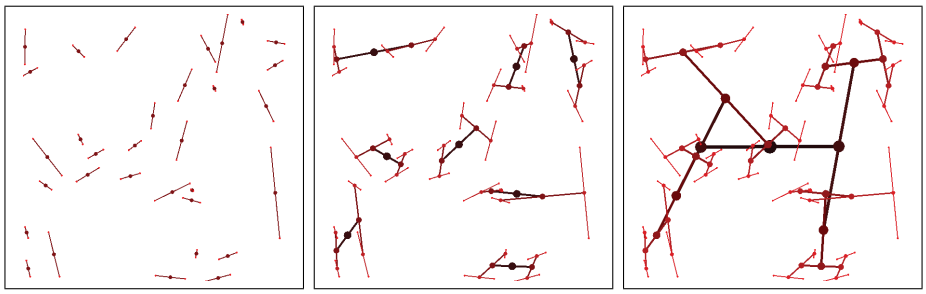


Figure 7.20: Example of matching and building the hierarchy

Due to the high complexity of the method, it is not practically possible to run it directly on large data. Clusters are first identified in the data. The matching is constructed within each cluster separately. Then a matching among particular clusters is constructed to merge the results together. The output is a binary tree that defines the space partitioning for the ray tracing. The implementation is capable of processing 5 million vertices in several hours. The proposed algorithm was published in [130]. The paper also documents a few unsuccessful attempts to solve the matching problem.

7.5.2 Experiments

The experiments discussed in this section were made on random data with uniform distribution. The program was written in C# for the .NET Framework 2.0. The testing computer was Pentium 4 3.2 GHz with 2 GB RAM, running Windows XP.

The time complexity of the Kuhn-Munkres algorithm is $\mathcal{O}(N^3)$. This was proven in practice as seen in Figure 7.21. There is no competitive program

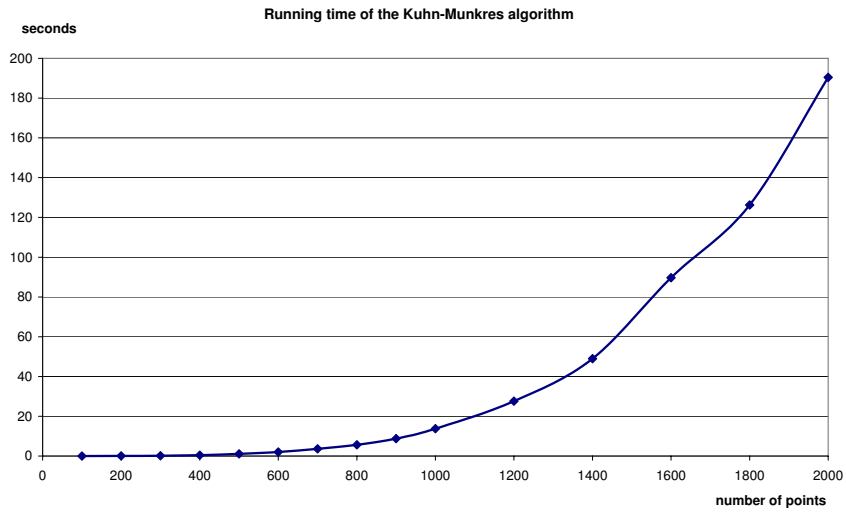


Figure 7.21: Running time of the Kuhn-Munkres algorithm

to compare with. The optimal Edmonds algorithm is what we wanted to avoid due to its complex implementation. There is no other method either, so a new approximate solution was developed which I believe is unique in this way.

Quite an unpleasant property of the Monte Carlo method is that it does not converge. Consecutive matchings generated by the algorithm have a rather random cost. The process does not converge to the minimal cost matching. The graph in Figure 7.22 shows the first 100 iterations of the Monte Carlo algorithm for a data with 100 vertices.

So it is hard to find a stopping condition when the solution is close enough to the optimum. Our experiments show that $\mathcal{O}(N)$ iterations should be done, assuming N is the number of input points. For practical applications, we recommend N or $10N$ iterations. The improvements in later iterations are less significant. The graphs in Figures 7.23 and 7.24 show the progress of improvements, i.e., decreases in the matching cost. The graphs show the results from three random datasets of 100 points after 10 000 iterations (blue, purple and brown line) and one dataset of 500 points after 50 000 iterations (green line). Figure 7.23 shows how the minimal known cost decreases over the iterations. Figure 7.24 shows the decrease in percent of

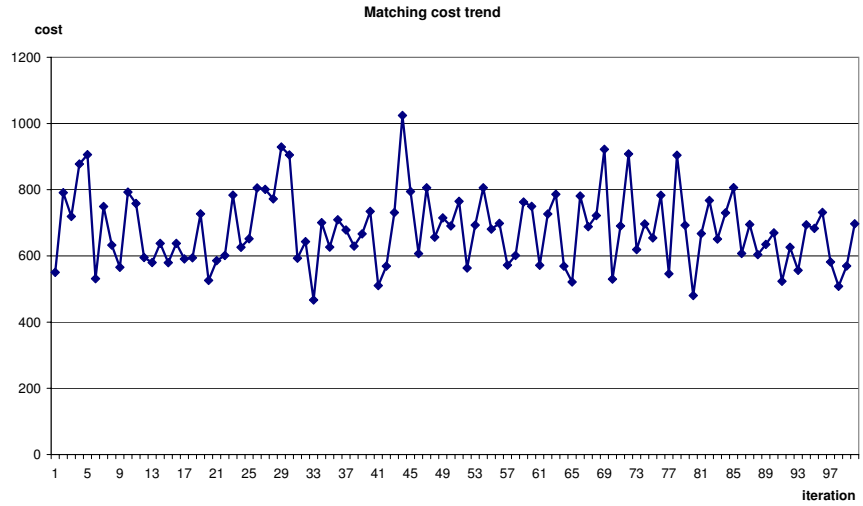


Figure 7.22: Trend of the matching cost over successive iterations of the Monte Carlo method

the previous minimal value. Please note the logarithmic scale on the x axis in both graphs. The number of iterations should be determined also with respect to the time we can spend by running the algorithm; the more the better.

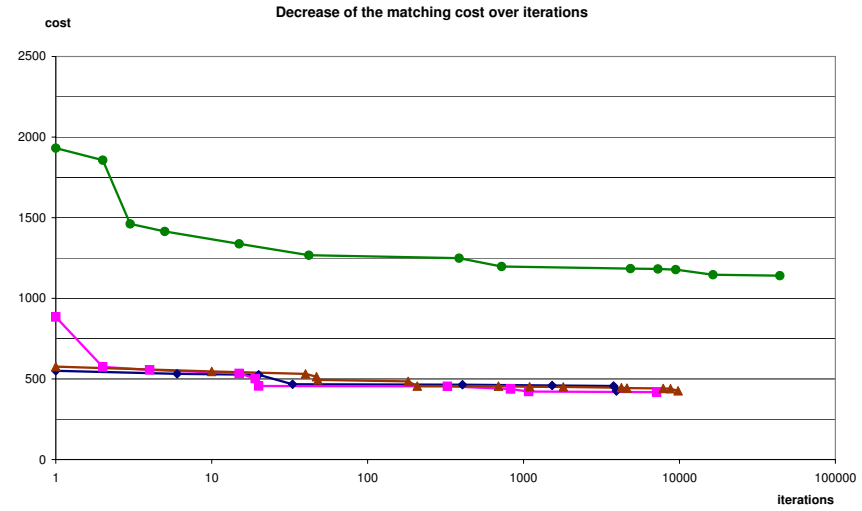


Figure 7.23: Trend of the minimal cost of matching over the iterations

To conclude, an alternative method to the optimal-but-complicated Edmonds algorithm for the minimal Euclidean matching was developed. The proposed method is relatively simple and finds an approximate solution to the matching.

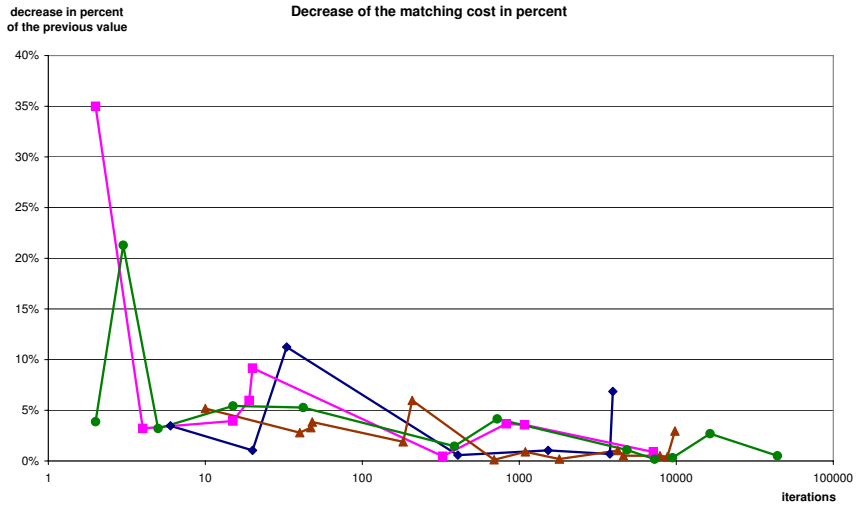


Figure 7.24: Decrease of the minimal cost expressed in percent of the previous minimum

7.6 Clustering on CUDA

In the pursuit of greater computation speed, the clustering was implemented on the GPU. Unfortunately, the local search algorithm uses non-trivial data structures so it is not the best candidate for the GPU. The single- / complete-link approaches, described in Section 4.2.1, were chosen for their simplicity. This section presents the computation of the clustering on the GPU using the CUDA framework. The results were published in [128].

7.6.1 The algorithm for the GPU

The complete-link algorithm starts with computing the distance matrix. Each element is computed by one GPU thread. Only half of the distances are actually computed, due to the symmetry of the distance matrix. Section 7.6.2 describes this in detail.

Each vertex is initialised as an individual cluster, and maintains a list of assigned vertices, which at the beginning contains only the vertex itself. All the rows of the distance matrix are *active*, meaning that the cluster corresponding to the row has not been merged into some other cluster. The algorithm then proceeds as described in Section 4.2.1.

Sequential algorithms mostly maintain a sorted list of the closest pairs of clusters to quickly find the closest ones to merge. Our GPU implementation pre-computes the mutual distances, but it does not sort them. It relies on the massive computing power and finds the closest clusters (the minimum in a distance matrix) by brute force. Maintaining the sorted list is inherently a sequential operation, and it would not fit the parallel computation.

To find the closest pair of clusters, each active row of the distance matrix is scanned by a GPU thread to find the minimum. Again, only half of the elements are scanned, due to the symmetry. The partial minima from the rows are gathered by the CPU to find the global minimum which identifies the closest clusters A and B.

The clusters are merged, specifically, B is merged into A. The list of vertices already assigned to B is copied to A. The row corresponding to cluster B in the distance matrix is *deactivated*.

Now the distances to the new cluster must be updated. For each active row of the distance matrix, except A (and the deactivated B), a GPU thread compares the distance to A with the distance to B. The greater value is kept as the distance to the merged cluster. The matrix symmetry ensures that the distances in the row corresponding to the merged cluster A will be updated as well.

7.6.2 Maintaining the distance matrix

The distance matrix used in the computation is indeed symmetric. The elements on the diagonal are of no use, so only the elements above the diagonal are really needed. This is, however, not well suitable for distributing the work among the parallel threads. If the matrix size is n, the first row contains $n - 1$ elements that need to be processed, while the last but one row contains just a single element to be processed.

A better load balancing can be elegantly achieved by a smart distribution of the matrix elements. We simply store $\lfloor \frac{n}{2} \rfloor$ elements in each row, starting from the first element to the right of the diagonal. If we get to the last column, we continue with the first column (count columns modulo n). The stored elements form a band above the diagonal and a triangle in the lower left corner. The idea is illustrated in Figure 7.25. The elements that are actually stored are marked grey. If n is even, $n/2$ elements will be stored twice (the dark grey elements in the figure), but this is no problem.

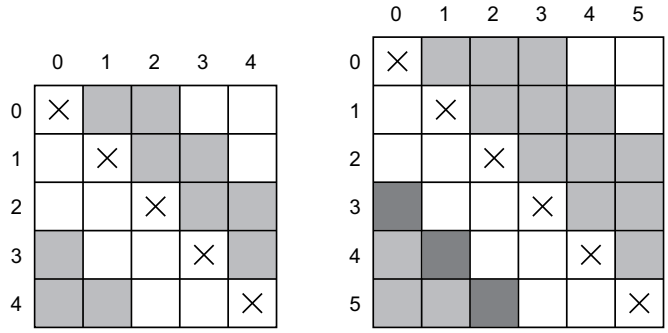


Figure 7.25: Scheme of the distance matrix storage for n odd and for n even, respectively.

The work for the parallel threads is then easily distributed as if the matrix elements would be stored in a rectangular matrix $n \times \lfloor \frac{n}{2} \rfloor$. Each thread is assigned a single row i' and processes the elements $j' \in \{0, 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$. Indices to the original $n \times n$ matrix are computed by the following indexing:

$$\begin{aligned} i &= i' \\ j &= (j' + i' + 1) \bmod n \end{aligned} \tag{7.9}$$

Converting the index back is more complicated:

$$\begin{aligned} &\text{if} \quad \left(0 < i - j < \frac{n+2}{2} \quad \text{OR} \quad j - i > \frac{n+2}{2}\right) \\ &\quad \text{swap}(i, j) \\ i' &= i \\ j' &= (j - i + n) \bmod n \end{aligned} \tag{7.10}$$

Fortunately, the backwards conversion is never needed in the algorithm.

7.6.3 Experiments

This section documents the experiments with the complete link clustering on the GPU using CUDA. The measurements were performed on Pentium 4 3.6 GHz processor and NVIDIA GeForce 8800 GTX graphics card. The GPU algorithm was compared with a similar implementation written in C running on the CPU. Several small 3D surface models were used as test data. Due to rather disappointing speedup, the development did not go on with larger data.

The results are summarised in Table 7.8. The overall speedup ranges from 15% to 40% which is not as great as expected. It is probably caused by fragments of work still being done on the CPU. Transferring the data between the main memory and the graphics card memory causes delays. The computation of the distance matrix is very fast already. The following steps of the complete link algorithm can be further optimised.

Table 7.8: Speedup of the CUDA implementation.

Dataset	Number of vertices	CPU time [s]	GPU time [s]	Speedup
Objects	1420	2.855	2.434	15%
Ellipsoid	2452	11.985	7.193	40%
Cow	2905	25.578	19.420	24%
Head CT	4098	70.369	54.951	22%

The conclusion is that it is not straightforward to design an efficient algorithm for the GPU. Getting a better speedup would require a dedicated research in this direction.

Chapter 8

Dynamic hierarchical triangulation

The dynamic hierarchical triangulation is the proposed tool for manipulating large geometric data. Dynamic means that points can be inserted and removed interactively at runtime. Hierarchical means that it uses the cluster hierarchy created by the data stream clustering described in Chapter 7. The hierarchy is stored on a hard disk, so there is no need to compute the time consuming clustering every time again. The solution presented in this thesis uses the Delaunay triangulation. Other type of triangulation can be used as well, as long as the construction algorithm allows inserting and deleting points. The proposed hierarchical triangulation can be constructed in either 2D, or in 3D where it would be more appropriate to call it a tetrahedralisation. Let us stay with the term triangulation for generality.

Initially, the triangulation of the highest level of the hierarchy is constructed. Each point at the highest level represents a cluster of points at a lower level. The clusters should have a reasonable size (about 100 to 1000 points) to partition the data into easily manageable pieces. It is then possible to interactively *expand* any of the clusters, i.e., insert all its points into the triangulation, and thus locally increase the level of detail. The expansion can continue down to the lowest level. Figure 7.8 on the page 69 shows an example of expanding clusters in the hierarchy (the triangulation is not shown for better readability). Nevertheless, we can expand only as many clusters as fit into the memory. It is therefore possible to *collapse* clusters that are no longer interesting, i.e., remove all the points and replace them by the cluster centre, and thus free memory for other data.

The triangles connecting the areas of a low and a high level of detail do not show any artefacts. When expanding or collapsing clusters, the popping effect may appear, i.e., an abrupt change in the model geometry can be observed. If this is unacceptable for the visualisation, it could be solved by geomorphs [57, 59]. It is a morphing technique proposed for the progressive

meshes to make a smooth animated transition when changing the level of detail. In our case, it is necessary to morph all the points of the cluster being expanded or collapsed.

The expansions and collapses can be controlled manually by the user – either by specifying the clusters to be expanded or collapsed one by one, or by defining a region of interest R where a high level of detail is desired. For visualisation, the region can be determined automatically according to the viewpoint. Clusters that are out of view are collapsed. If the available memory is running low so that further collapsing is necessary, the clusters farthest from R should be collapsed first. Other applications might require a different approach. For instance a computing task, such as a height field interpolation, would control the level of detail according to an error measure.

Figure 8.1 shows a 2D triangulation of the digital elevation map of the whole world [139] consisting of 310 million vertices. Most of the area is triangulated using the second level containing just 3600 vertices. The western and the central Europe is expanded to the first level. The central part of Denmark is expanded down to the level zero – the full resolution. Triangulations at particular levels are rendered separately in different colours for visual clearness. Of course, our method can also work with all the points in a single triangulation.

Figure 8.2 shows a 3D Delaunay tetrahedralisation of the Lucy statue [132]. The whole model consists of 10 million vertices. The figure shows the triangulation of the second level containing just 1825 vertices. No clusters were expanded because the image is already cluttered. A nicer visualisation can be achieved by restoring the non-convex shape as described in Section 8.3.

The 2D solution was published in [127], the 3D solution including the non-convex shape restoration is submitted for publication [129].

8.1 Cluster expansion and collapse in 2D

Expansion of a cluster is not difficult. All its points are loaded from the hard disk. The file format is described in Section 7.2. The points are then simply inserted one by one into the triangulation using the well known incremental insertion algorithm [54, 92, 138]. I believed it could be possible to insert a pre-triangulated cluster all at once. However, this has not proved to be profitable because clusters may overlap and it would be necessary to modify the pre-constructed triangulations. Further research could be made in this direction later.

Collapsing a cluster deserves a more detailed explanation. The goal is to remove a set (cluster) of points S from the triangulation. The algorithm is based on the technique of removing a vertex from the triangulation [138]. The original technique removes a vertex along with all incident triangles.

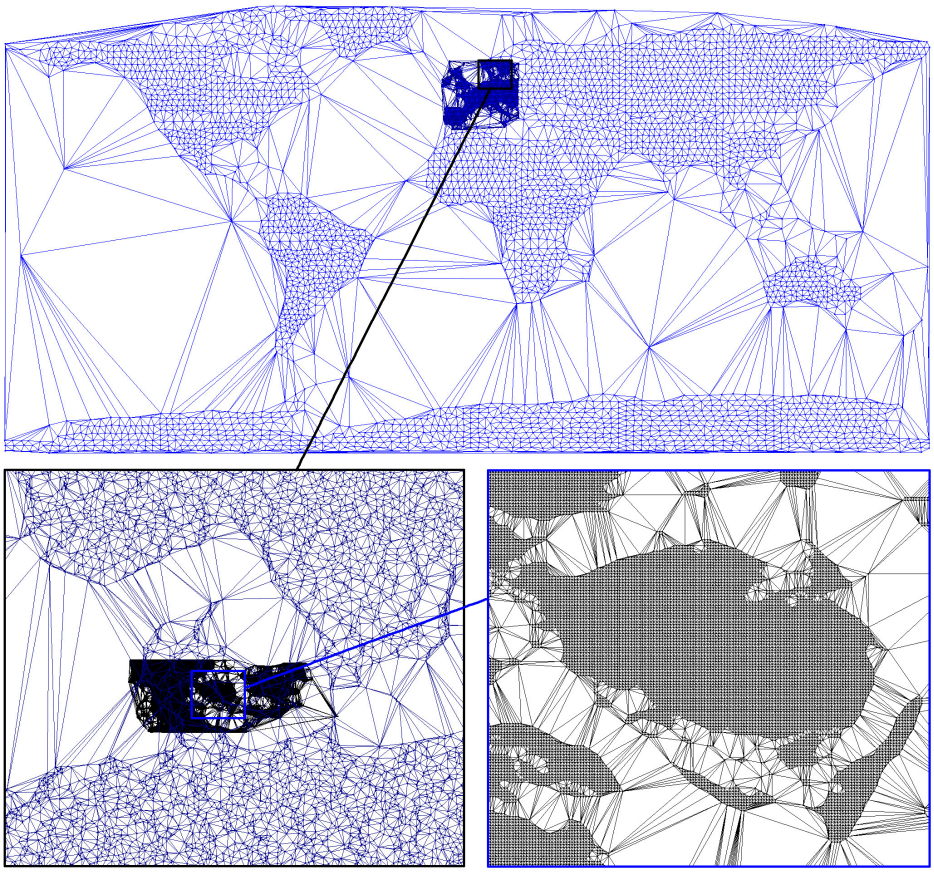


Figure 8.1: Hierarchical triangulation of the elevation map of the whole world. Data provided by [139].

The resulting *hole* is then re-triangulated. The algorithm proposed here extends the technique by further enlarging the hole by removing additional vertices of S before the re-triangulation. The solution is simple, reliable, and runs relatively fast. The algorithm works as follows:

1. Create a hole – remove the first vertex.
2. Enlarge the hole – continue with removing further vertices as described below
3. Re-triangulate the hole.
4. If not all cluster vertices have been removed, go to 1.

Figure 8.3 shows a sample situation after removing three points. The important condition is that a point can be removed only if it lies on the hole boundary, so that removing the point will enlarge the hole. Removing a point from elsewhere will create a new hole. Such holes could later touch

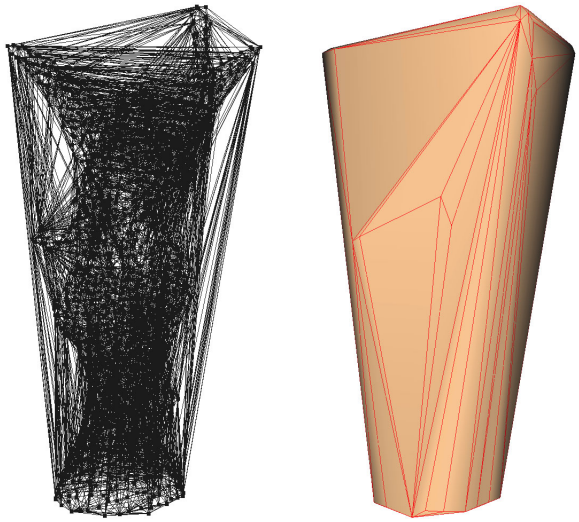


Figure 8.2: Tetrahedralisation of Lucy, wire frame and solid rendering. Data provided by [132].

each other or merge together. That would require complicated overhead, and it is unlikely to bring a major improvement to the algorithm. When a vertex appears more than once at the hole boundary, see examples in Figure 8.4, it is a degenerate case which is disallowed for the same reason as multiple holes are. If any vertex removal should result in such a degeneracy, the removal is cancelled.

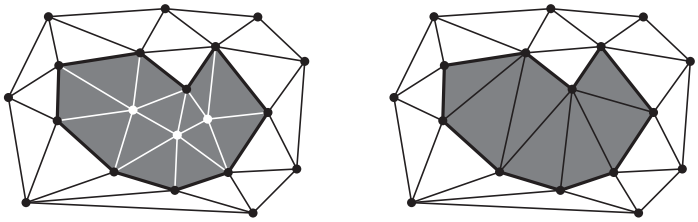


Figure 8.3: Hole after removing three vertices and after the re-triangulation. The hole is dark grey, removed edges and vertices are white.

When no further vertex can be removed, the hole is re-triangulated by a simple *ear cutting algorithm*. An *ear* is generally a valid triangle formed by three subsequent vertices v_i, v_{i+1}, v_{i+2} at the hole boundary. *Cutting an ear* means adding the triangle $v_i v_{i+1} v_{i+2}$ to the triangulation and removing the second vertex v_{i+1} from the boundary. The implementation goes around the hole boundary and it tests ears whether they fulfil the Delaunay property. Only the points at the boundary are included in the test. Other points in the triangulation do not play any role. Ears that pass the test are cut.

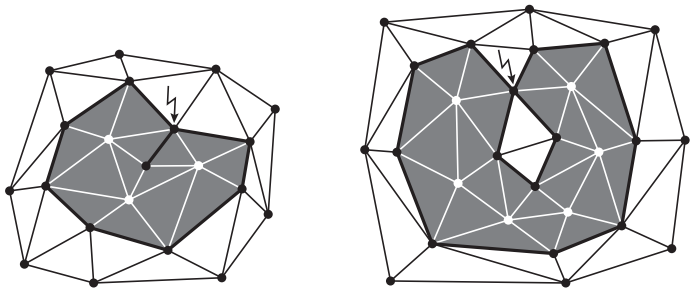


Figure 8.4: Examples of degenerate holes

8.2 Cluster expansion and collapse in 3D

The proposed 3D solution uses the triangulation algorithm developed by my colleague Michal Zemek [144]. Expansion of a cluster is practically the same as in 2D. To insert a new point, the tetrahedron containing the point is located by the walking algorithm [29]. The tetrahedron is split into four new tetrahedra by connecting its vertices to the point being inserted. Tetrahedra are then checked and eventually flipped to restore a proper Delaunay tetrahedralisation.

To collapse a cluster, we need to remove a set (cluster) of points S from the triangulation. The points are removed one by one using an algorithm based on [27] and [30]. Let p be the vertex to be removed. All the tetrahedra incident to p are removed creating a cavity in the triangulation. The triangular faces of the cavity form a star-shaped polyhedron denoted as $\mathcal{P}(p)$. The polyhedron $\mathcal{P}(p)$ is re-triangulated by the ear cutting algorithm. An *ear* ε of the polyhedron $\mathcal{P}(p)$ is a set of 2 or 3 neighbouring faces of $\mathcal{P}(p)$ such that the tetrahedron determined by these faces lies inside $\mathcal{P}(p)$, see Figure 8.5. Note that each ear is given by exactly four vertices of $\mathcal{P}(p)$. Cutting off the ear ε means creating its corresponding tetrahedron and connecting it to the triangulation outside $\mathcal{P}(p)$. The so called *power* [27] is computed for each ear. It is a quotient of the in-sphere test of p with respect to the ear ε, and the 3D orientation test of the vertices of ε. All the ears are stored in a priority queue. The ears minimising the power are being removed from the queue and cut off until the whole cavity is re-triangulated. This leads to the time complexity of $O(k_R \log k_N)$, where k_R is the number of tetrahedra created to re-triangulate $\mathcal{P}(p)$, and k_N is the initial number of vertices of $\mathcal{P}(p)$.

8.3 Restoring the non-convex shape in 3D

As seen in Figure 8.2, the Delaunay triangulation essentially fills the whole convex hull of the data. Rendering the result shows just a formless shape filled with tetrahedra which can be confusing. Therefore, a post-processing

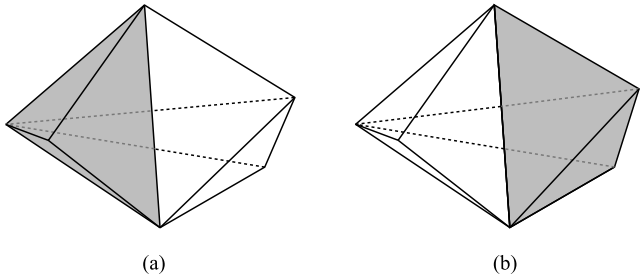


Figure 8.5: Ears (shaded) of polyhedron $\mathcal{P}(p)$; (a) an ear given by 3 faces of the polyhedron, (b) an ear given by 2 faces of the polyhedron.

of the triangulation was proposed to restore the original shape. The term "reconstruction" is intentionally avoided, because the algorithm is not supposed to be a rigorous surface reconstruction. It can be easily replaced by a more elaborate algorithm if needed.

Figure 8.6 shows the hierarchical triangulation of the new David model [132]. The whole model contains 468.6 million points. The non-convex shape of the statue was constructed using the algorithm discussed in this section. Level 3 of the hierarchy containing just 227 vertices provides a very coarse model of the whole statue. The facial part is refined by adding 1490 more vertices from the level 2. The eyes have even bigger resolution thanks to another 3084 vertices from level 1.

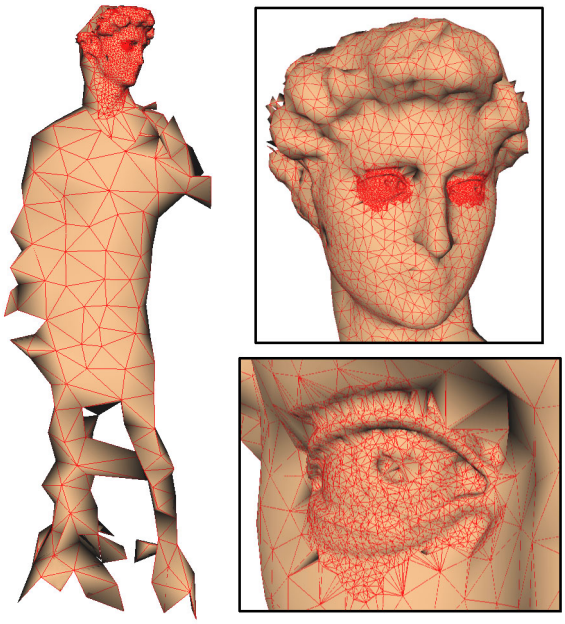


Figure 8.6: Hierarchical triangulation of the new scan of David. Data provided by [132].

We would like to keep only the tetrahedra that constitute the inside of the original model, and remove the other ones. The first idea would be to distinguish the inner tetrahedra from the outer ones according to their shape. This is usually easy in a 2D triangulation because the vertices fill the interior of the triangulated shape. The problem with 3D data scanned from the surface of a model is that the triangulation has poorly shaped tetrahedra both outside and inside the model. It is not possible to evaluate the tetrahedra independently. However, the vertices on the model surface are relatively close together, so the triangular faces forming the model boundary – call them *boundary faces* – are likely to be small. This is a heuristic to approximately distinguish the exterior and interior tetrahedra. See an example in Figure 8.7 (only 2D for clarity). It shows a triangulation of a subset of the famous bunny [132]. The boundary edges are drawn thicker.

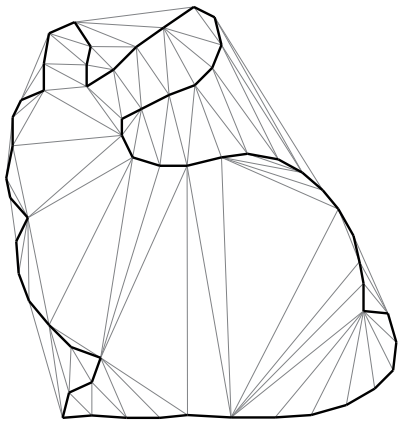


Figure 8.7: Distinction of boundary faces

The idea is to successively remove tetrahedra, starting from the convex hull, and going deeper to the model until its boundary is reached. Except the surface, the algorithm also preserves the interior tetrahedra of the model, so they could be used for further computational processing. To describe the algorithm more specifically, we need to introduce the term *external face*. A tetrahedron has an external face if there is no neighbouring tetrahedron sharing that face. At the beginning, the external faces are the faces of the convex hull. As tetrahedra are removed, further faces will become external.

There are several possibilities how to decide whether an external face is boundary or not. They are based on the premise that a boundary face is small compared to other faces of the tetrahedra. This thesis suggests two methods for finding the boundary faces.

The simplest solution is to measure edge lengths. Let ABC be an external face of the tetrahedron $ABCD$. Let d_{ABC} be the perimeter of ABC, and let d_{other} be the sum of the lengths of the other three edges of $ABCD$. If $d_{ABC} < 0.5 \cdot d_{\text{other}}$, then ABC is declared a boundary face. The mul-

tiplicative constant was derived from experiments and it can be adjusted according to a particular model. Although this metric works, it makes holes in the model too often.

A better solution is based on the Gabriel property [9]. Consider the minimum containment sphere of the face ABC, i.e., the smallest sphere containing ABC, see Figure 8.8 (a) for illustration. Note that not all three vertices must always lie on the sphere. There may be two vertices on the sphere and the third vertex inside as seen in Figure 8.8 (b). If the sphere does not contain the fourth vertex D, then ABC satisfies the Gabriel property, and ABC is declared a boundary face.

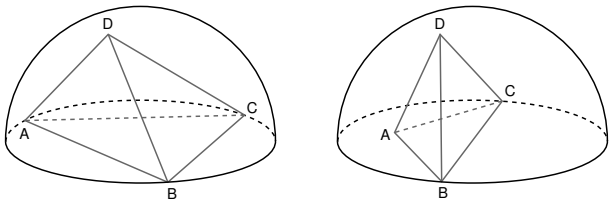


Figure 8.8: Illustration of the minimum containment spheres for the Gabriel property; (a) point A lies on the sphere, (b) point A is inside the sphere.

The complete algorithm to restore the non-convex shape is described as follows:

```
Make a queue of the tetrahedra that have at least one external
face.

While (queue not empty)
{
  Take out a tetrahedron from the queue.

  If (none of its external faces is on the model boundary)
  {
    Remove the tetrahedron from the triangulation.

    Put all its neighbours into the queue (these neighbours
    now have an external face).
  }
  else
    Keep the tetrahedron in the triangulation.
}
```

Figure 8.9 shows an example of restoring the non-convex shape of the Lucy model [132]. Most of the model is triangulated using the second level containing 1825 vertices. The bottom part is shattered, because there are too few vertices at the base, and the algorithm is too simple to handle that

correctly. The right hand is expanded to the first level, adding 1655 more vertices. The index finger tip is expanded down to the level zero – the full resolution – adding another 2501 vertices.

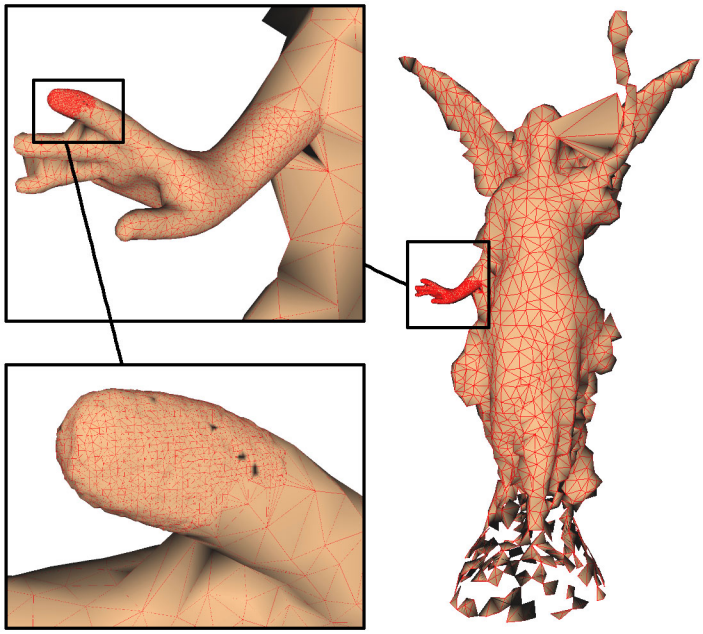


Figure 8.9: Hierarchical triangulation of Lucy with the non-convex shape restored

8.4 Experiments

This section evaluates the dynamic hierarchical triangulation. The proposed algorithms were implemented in C# in the .NET Framework 2.0. Experiments were done on Intel Pentium 4 3.2 GHz with 2 GB RAM and SATA HDD, running Windows XP.

8.4.1 Comparison to alternative approaches

From a general point of view, out-of-core and hierarchical algorithms can process large data, but they are not endlessly scalable. The solution proposed in this thesis is based on the data stream concept, so it can easily handle really gigantic data. Most streaming algorithms used in computer graphics require at least partially ordered data or they need a large buffer for reordering the incoming data. Sometimes the algorithms even make multiple passes over the stream. The proposed method makes only a single pass, and it can handle potentially shuffled data.

To the best of my knowledge there is no directly competitive algorithm. Let me discuss how the proposed method compares to existing related tech-

niques. The progressive meshes for large models [112] need an already triangulated data which is a problem itself. The concept of streaming meshes [66] allows processing a huge triangular mesh in a small memory. It needs approximately sorted data, and it does not provide any hierarchy for handling the mesh in a lower resolution or working with only a part of the data.

There are streaming algorithms for large point clouds [110, 6]. They are designed for local computations on the points such as the normal estimation. Several operations can be chained one after another. The input data must be sorted along one axis, and it must be processed as the whole stream; there is no mechanism to access only a part of the data or to get a lower resolution. The output is still a point cloud without additional structure.

Point based rendering techniques [117, 141, 47] are capable of visualising huge point data in real time. They are focused on producing the image output, and they do not support further computations on the data.

Perhaps the closest method is the data stream Delaunay triangulation by Isenburg et al. [70]. Its clear advantage is the running time. The authors processed 500 million points in 48 minutes and 4.5 billion points in less than 7 hours (on a slightly weaker hardware than the one used for the experiments in this thesis), even though the algorithm makes 3 passes over the stream. The data has to be at least approximately sorted, nevertheless, the authors correctly remark that real data usually is naturally sorted. The result is one huge triangulation of the whole dataset.

The solution proposed in this thesis is notably slower; it needs 2 hours to process 470 million points. A significant advantage is that it makes only a single pass over the data, and it does not need any sorting. The greatest benefit is the flexibility. The data stream clustering builds the hierarchy of clusters, and stores it on the hard disk. The data can be then accessed efficiently in different resolutions. The created multiresolution model allows us to select different level of detail in various parts of the model.

The dynamic 3D triangulation is constructed in real time. A cluster of about 100 points is expanded practically immediately, collapsing it takes less than a second. The triangulation constitutes a multiresolution model with an interactive control over the level of detail.

The cluster hierarchy can also be used for various other purposes, especially as a space partitioning for accelerating computations on large data. It is used for studying protein molecules [145]. We are currently developing the cluster hierarchy for geodesic applications.

8.4.2 Point Removal Speedup

To collapse a cluster it is necessary to remove all its points from the triangulation. Instead of the traditional algorithm which removes the points one by one, a new technique was implemented that removes multiple points at a time. The proposed method was described in Section 8.1.

The experiments were performed on the Grand Canyon and the Earth datasets with some random clusters expanded and then collapsed. Table 8.1 summarises the experiments. The *Overhead* column lists the times necessary for the maintenance of the vertex array and passing the triangulation data structure for displaying. These times are the same for both methods – the conventional approach to removing points (denoted *One by one*) and the newly proposed technique (the *Hole enlarging*). The total time the user needs to wait for collapsing the clusters is the time for removing the points plus the overhead.

Table 8.1: Time needed for collapsing the clusters, i.e., removing their points from the triangulation

Clusters collapsed	Points removed	Overhead [seconds]	One by one [seconds]	Hole enlarging [seconds]	Speedup
15	4 454	0.01	0.09	0.09	0 %
30	9 415	0.08	0.23	0.17	26 %
60	19 095	0.35	0.49	0.33	33 %
125	36 690	1.37	1.56	0.72	54 %
250	73 060	6.66	4.04	1.54	62 %
500	146 741	35.23	10.34	5.54	46 %

The results partially depend on which clusters are being collapsed (their size and relative position). The table shows the average results. There is not much difference for a low number of clusters (containing fewer than 5000 points in total) since the collapsing is almost instantaneous. Nevertheless, for 125 clusters and more, the speedup reaches approximately 50 % which shows a good efficiency. The maintenance of the data structures (the *Overhead* column) is currently the bottleneck.

Chapter 9

Conclusion

The principles related to acquisition, manipulation, storage, and visualisation of large geometric data were discussed in the first part. It surveyed large data in general and introduced the data stream approach for processing huge datasets efficiently. Selected clustering techniques were described, paying special attention to the algorithms for data streams. The triangulation was described for giving structure to unordered point clouds. Specifically the Delaunay triangulation was discussed with emphasis on large data.

The second part presents the contributions to the current state of the art in manipulating large geometric data. A novel approach to handling huge data was proposed, concentrating mainly on point data. It consists of two stages. The data is first preprocessed by the hierarchical data stream clustering. The one-pass streaming technique allows processing huge datasets efficiently. The resulting hierarchy of clusters constitutes a multiresolution model of the original data. Various parts of the model can be efficiently accessed in different resolutions. The clustering organises the data according to various, even nongeometric, criteria, providing a high versatility. Building the cluster hierarchy is a time consuming process. It can take a few hours for half a billion points. The hierarchy is stored on the hard disk, so the preprocessing is done only once.

The second stage uses the cluster hierarchy to construct a dynamic hierarchical triangulation. It presents a rough shape of the whole data in a low resolution using the top level of the hierarchy. The level of detail can be increased on demand in specific parts of the model by inserting additional points from a lower level of the hierarchy. It is possible to get to the full resolution of the original data, but only in a limited area governed by the available memory. The level of detail can be later reverted back to the coarse model to free memory for refining other areas. All this works interactively in real time.

A salient open problem is the processing of entities more complex than points. Triangle soup has been processed for a ray tracing application. Ex-

periments are being done with polylines constituting a road network. Except the geometry, the entities may have additional nongeometric attributes. One of the major challenges is the interpretation of such attributes to quantify the similarity between the entities, and to do the clustering.

The automatic cluster expansion/collapse could be done better. The current solution expands the clusters that are close to the viewer or close to the centre of the view. A more clever algorithm could take the model silhouette into account. Another interesting problem is to adapt the proposed solution for publishing large data on the Internet.

Appendix A

More examples of the results

Several more examples of the multiresolution models created by the proposed solution are shown here.

A.1 Cluster hierarchy of geographic data

Figures A.1–A.3 show the multiresolution model of a part of the road in the Czech village Běstvina. The dataset was kindly provided by GEOVAP, Czech Republic [44]. Only points are rendered for readability. The data contain 17.3 million laser scanned points. The figures display all the 309 points (pink) from the second level of the hierarchy. Additionally, there is a subset of 8 586 points (red) from the first level, and a subset of 42 020 points (maroon) from the full resolution. Figure A.1 presents the approximate course of the road, Figure A.2 is a close-up of a part of the road with surrounding trees and bushes, Figure A.3 is a detail of the ditch along the road with a bush raising on the right.

Figures A.4–A.9 show the multiresolution model of a 10 km fragment of the Czech motorway D11. The dataset was kindly provided by GEOVAP, Czech Republic [44]. The data contain 468.2 million laser scanned points. The figures display all the 202 points (big maroon) from the third level of the hierarchy. There are subsets of 2 428 points (pink) from the second level, 704 points (red) from the first level, and 2 122 points (small maroon) from the full resolution.

Figure A.4 is a collage of the subsequent images to better understand the close-ups from slightly different angles. Figure A.5 gives an overview of the fragment of the motorway, Figure A.6 is a close-up of the motorway junction with the grade separation, Figure A.7 shows the two pillars supporting the bridge, Figure A.8 is a close-up of the pillars, Figure A.9 is a detail of the left pillar where you can clearly see the scan lines of the recording device.

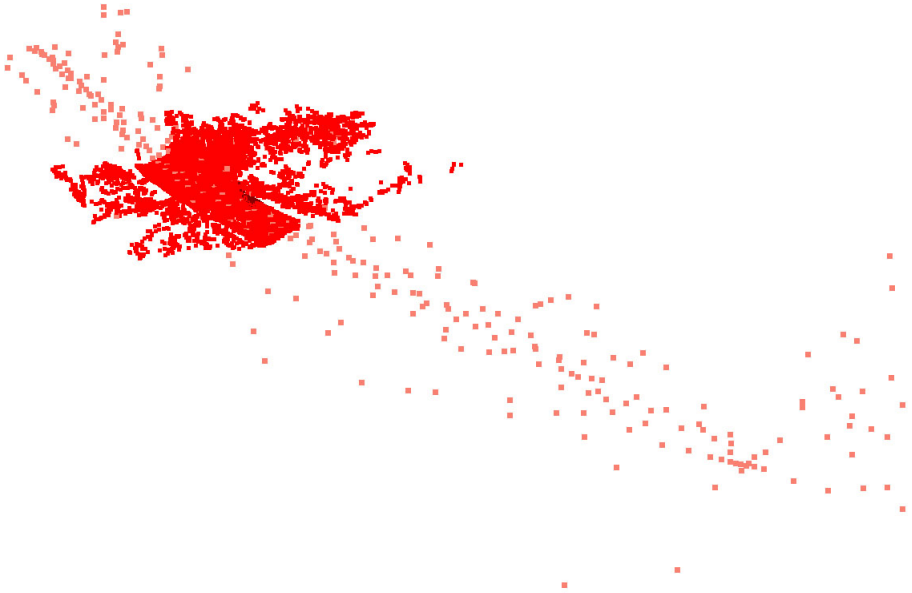


Figure A.1: Overall view of the road in Běstvina

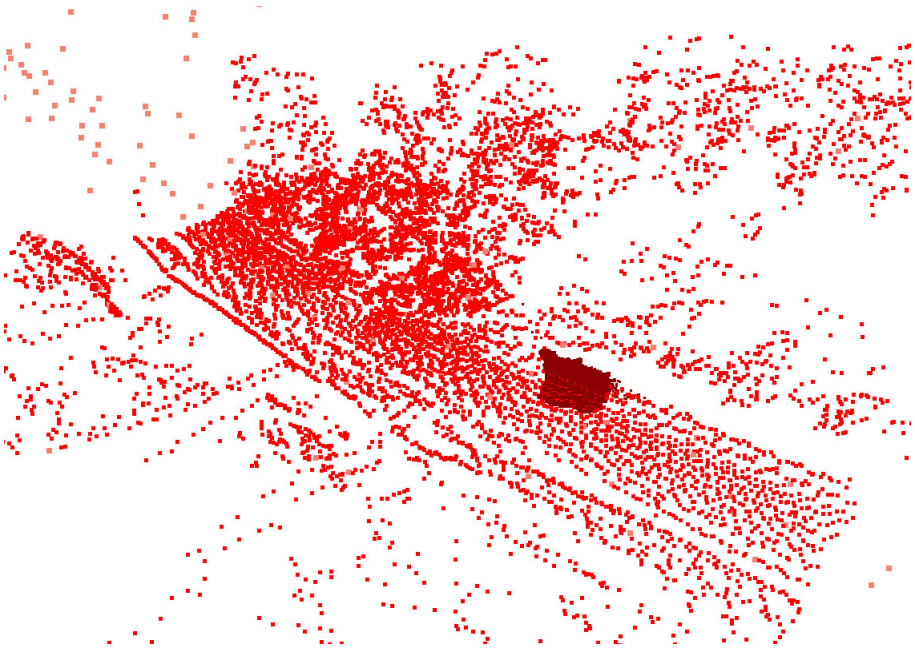


Figure A.2: Close-up of a part of the road

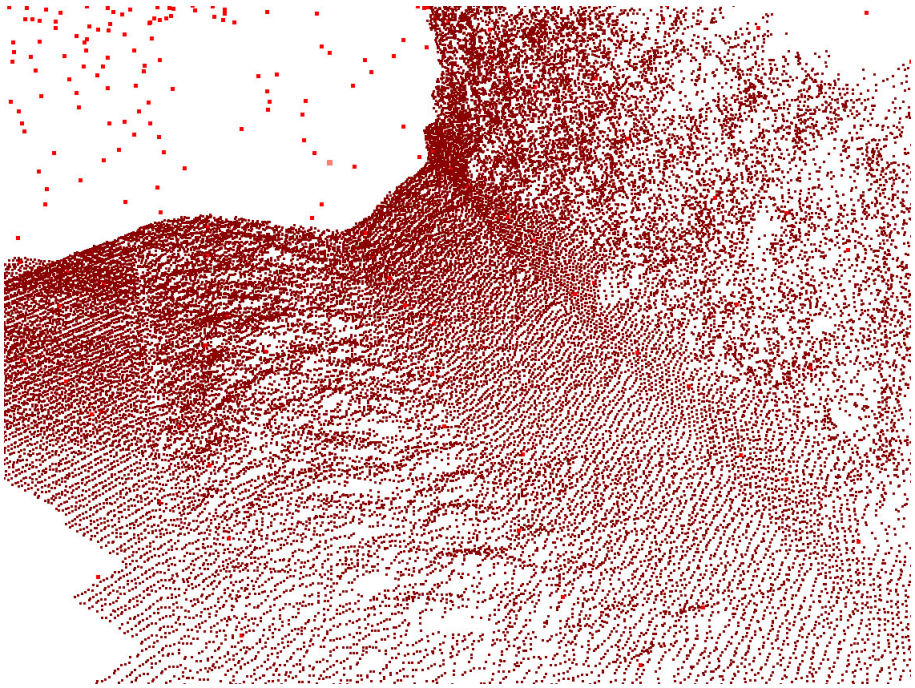


Figure A.3: Detail of the ditch along the road

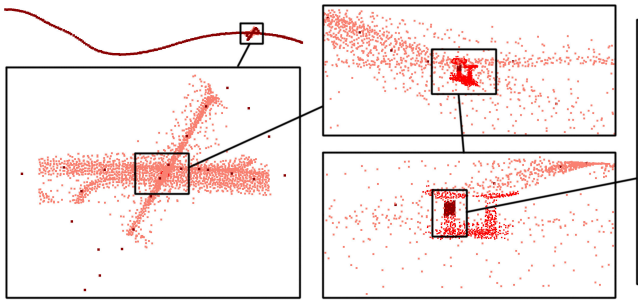


Figure A.4: Collage of the subsequent images of D11

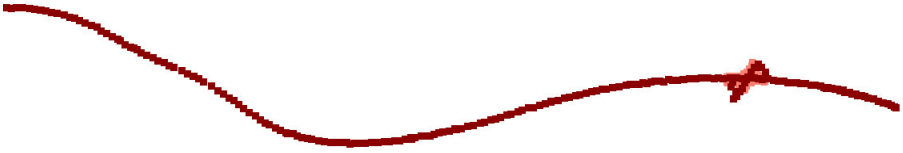


Figure A.5: Overview of the motorway D11

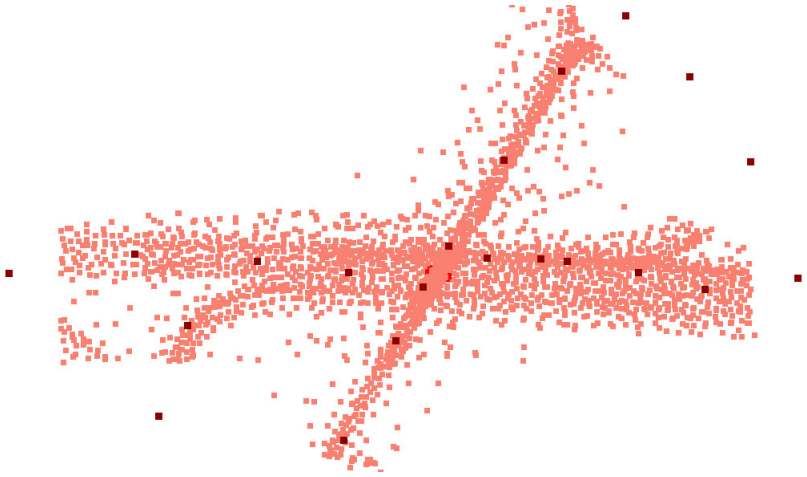


Figure A.6: Close-up of the motorway junction

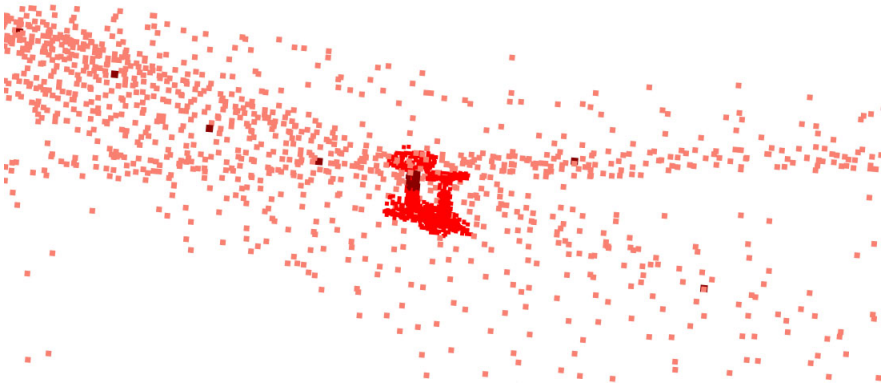


Figure A.7: Two pillars supporting the bridge

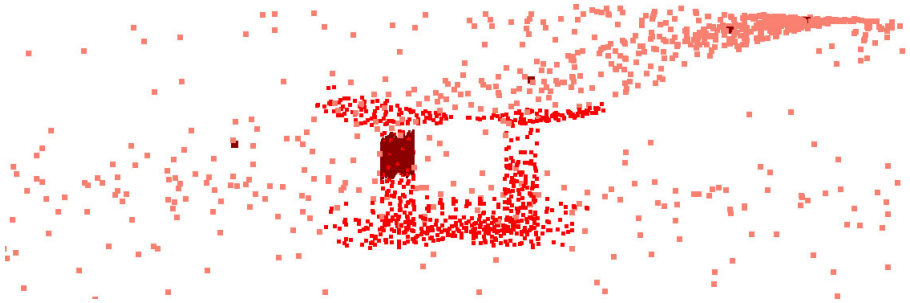


Figure A.8: Close-up of the pillars

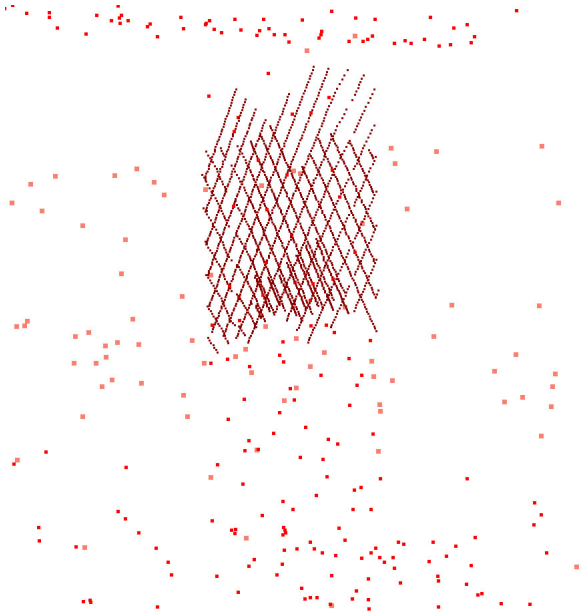


Figure A.9: Detail of the left pillar

A.2 Hierarchical triangulation of Michelangelo's statues

Figures A.10–A.12 show the multiresolution 3D triangulation of the statue of David [132] scanned in 2011. The input data contain 468.6 million points. Most of the triangulation was constructed from the 227 points of the third level, 1 490 more points from the second level were added to the head, and additional 3 084 points from the first level were added to the eyes. Figure A.10 gives an overall view, Figure A.11 is a close-up of the head, Figure A.12 is a detail of the face.

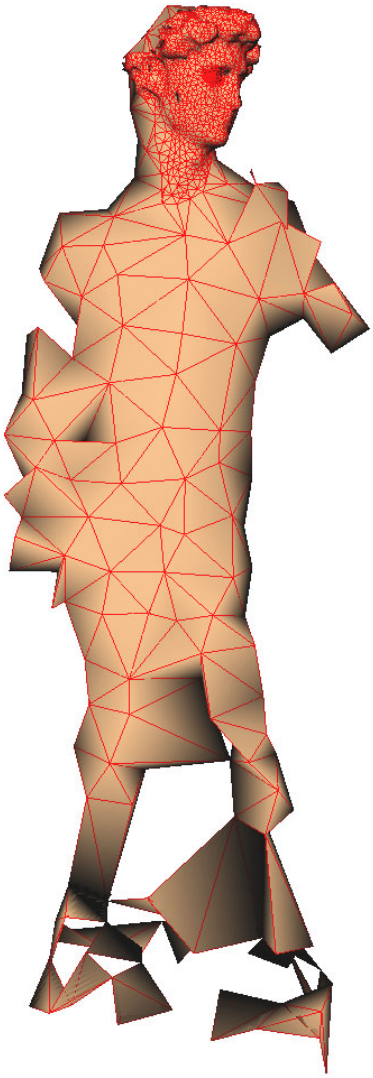


Figure A.10: Overall view of the David

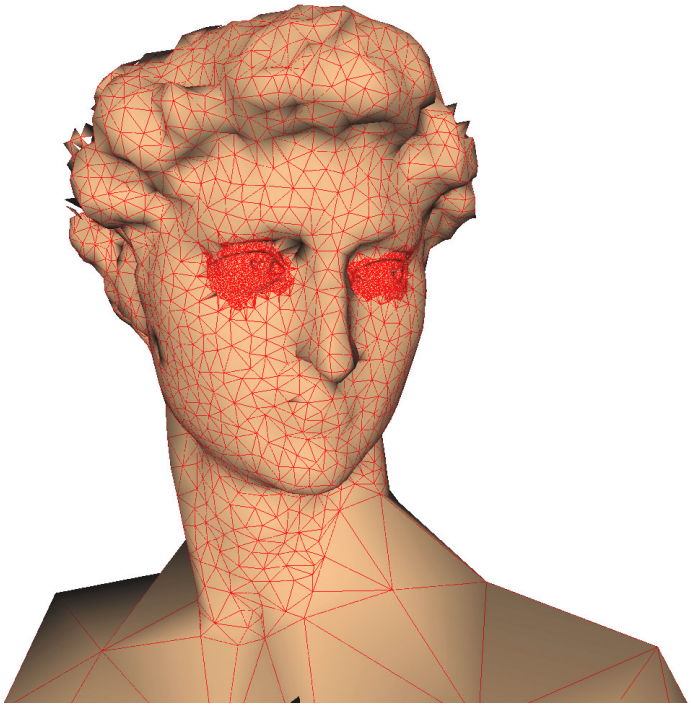


Figure A.11: Close-up of the head

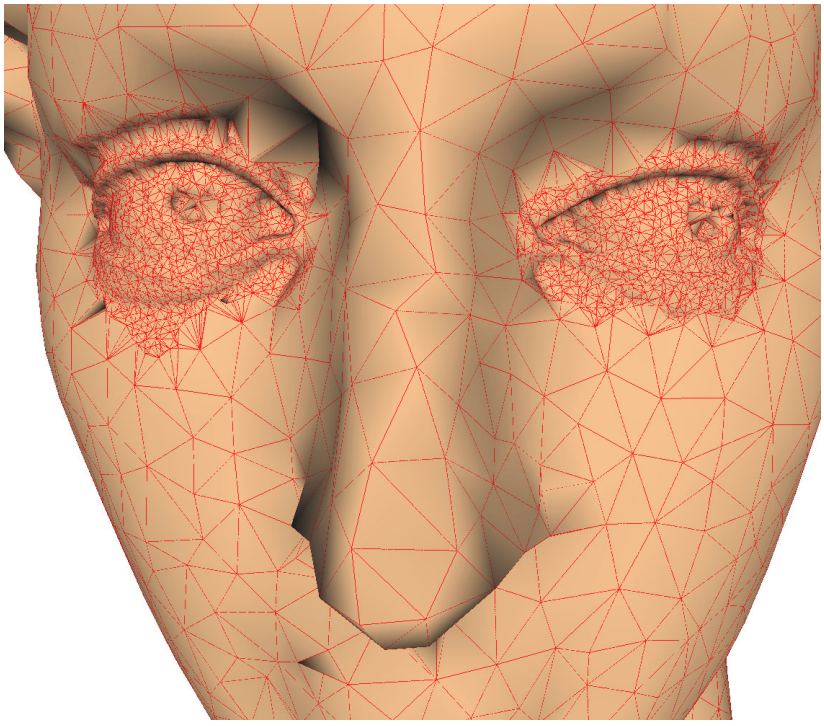


Figure A.12: Detail of the face

Appendix B

Activities

B.1 Publications

J. Skála and I. Kolingerová. One-pass hierarchical clustering and dynamic 3D triangulation of huge streaming data, 2012. *Submitted for publication in Graphical models, Elsevier.*

J. Skála and I. Kolingerová. Dynamic hierarchical triangulation of a clustered data stream. *Computers & geosciences*, 37(8):1092-1101, Elsevier, 2011.

J. Skála and I. Kolingerová. Faster facility location and hierarchical clustering. *International journal of computers*, 5(1):132-139, NAUN, 2011.

> Cited by:
>
> Z. Prokopová, P. Šilhavý, and R. Šilhavý. Preview of methods and tools for operating data analysis. *International journal of mathematical models and methods in applied sciences*, 5(6):1102–1109, NAUN, 2011.

J. Skála and I. Kolingerová. Accelerating the local search algorithm for the facility location. In *WSEAS MACMESE 2010*, pages 98-103, 2010.

J. Skála. Algorithms for manipulation with large geometric and graphic data. Technical report No. DCSE/TR-2009-02, Dept. of computer science and engineering, University of West Bohemia, 2009.

J. Skála, I. Kolingerová, and J. Hyka. A Monte Carlo solution to the minimal Euclidean matching. In *ALGORITMY 2009*, pages 402–411, 2009.

J. Skála. Data stream hierarchical clustering library, 2009. *Software* `http://www.kiv.zcu.cz/index.php?id=557&produkt_id=38`

M. Zemek, J. Skála, I. Kolingerová, P. Medek, and J. Sochor. Fast method for computation of channels in dynamic proteins. In *Vision, modeling, and visualization 2008*, pages 333–342, 2008.

Cited by:

D. Obrul, Y. Liu, and B. Žalik. Progressive visualization of losslessly compressed DICOM files over the Internet. Journal of medical systems, 36(3):1927–1933, Springer, 2012.

J. Skála and I. Kolingerová. Clustering geometric data streams. In *SIGRAD 2007*, pages 17–23, 2007.

Cited by:

M. Zemek. Regular triangulation in 3D and its applications. Technical report No. DCSE/TR-2009-03, Dept. of computer science and engineering, University of West Bohemia, 2009.

J. Rus and L. Váša. Analysing the influence of vertex clustering on PCA-based dynamic mesh compression. In *Articulated motion and deformable objects*, pages 55–66, Springer, 2010.

B.2 Significant scientific talks

Dynamická hierarchická triangulace velkých dat. At VŠB – Technical university of Ostrava, 2009

Clusterování data streamů a hierarchická triangulace. At VŠB – Technical university of Ostrava, 2008.

Hierarchical clustering of large geometric data. At University of Maribor, Slovenia, 2007.

Clustering geometric data streams, poster. At MADALGO Summer school 2007 on data stream algorithms, Århus, Denmark, 2007.

B.3 Participation in projects

Czech science foundation (GAČR), project 201/09/0097
Research and development of algorithms for large data in computer graphics. 2009–2012

EU project CZ.1.07/2.3.00/09.0050 (SPAV)
Organising scientific seminars. 2009–2012

Ministry of education, youth and sports, project LC 06008 (CPG)
Research and development of algorithms for modelling, visualisation, and interaction in virtual reality. 2006–2011

Ministry of education, youth and sports, project 2C 06002 (VIRTUAL)
Research of geometrical computations in homogeneous space. 2006–2007

Ministry of education, youth and sports, project KONTAKT 5/2005-06
Research of computational geometry in cooperation with University of Maribor, Slovenia. 2006

Bibliography

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1):293–327, 1988.

[2] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *ACM Symposium on Computational Geometry*, pages 211–219, 2003.

[3] G. Ball and D. Hall. ISODATA: A novel method of data analysis and pattern classification. Technical report, Stanford Research Institute, Menlo Park, 1965.

[4] A. Baraldi and P. Blonda. A survey of fuzzy clustering algorithms for pattern recognition – part II. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(6):786–801, 1999.

[5] J. Bezdek and R. Ehrlich. FCM: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2):191–203, 1984.

[6] J. Bösch and R. Pajarola. Flexible configurable stream processing of point data. In *International Conference on Computer Graphics, Visualization and Computer Vision (WSCG)*, pages 49–56, 2009.

[7] T. Boubekeur and M. Alexa. Mesh simplification by stochastic sampling and topological clustering. *Computers & Graphics*, 33(3):241–249, 2009.

[8] K. Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223–228, 1979.

[9] R. Chaine. A geometric convection approach of 3-D reconstruction. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 218–229, 2003.

[10] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 17(1):78–86, 1970.

[11] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Automata, Languages and Programming*, 2380:784–794, 2002.

[12] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *IEEE Symposium on Foundations of Computer Science*, pages 378–388, 1999.

[13] M. Charikar, S. Khuller, D. M. Mount, and G. Narasimhan. Algorithms for facility location problems with outliers. In *Symposium on Discrete Algorithms*, pages 642–651, 2001.

[14] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 436–447, New York, NY, USA, 1998. ACM.

[15] F. A. Chudak. Improved approximation algorithms for uncapacitated facility location. *Lecture Notes in Computer Science*, 1412:180–194, 1998.

[16] P. Cignoni, C. Montani, and R. Scopigno. DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. *Computer Aided Design*, 30:333–342, 1998.

[17] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[18] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. *Computer Graphics*, 30(Annual Conference Series):119–128, 1996.

[19] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using Hamming norms (How to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003.

[20] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *LATIN 2004: Theoretical Informatics*, pages 29–38, 2004.

[21] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30(1):249–278, 2005.

[22] The Crater Lake data set. Created by M. Garland. `http://mgarland.org/dist/models/crater.smf.gz`.

[23] E. Danovaro, L. De Floriani, E. Puppo, and H. Samet. Multi-resolution out-of-core modeling of terrain and teological data. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 143–152, New York, NY, USA, 2005. ACM.

[24] L. De Floriani, E. Puppo, and P. Magillo. A formal approach to multi-resolution modeling. *Geometric Modeling: Theory and Practice*, pages 302–323, 1997.

[25] B. N. Delaunay. Sur la sphère vide. *Izvestia Akademia Nauk SSSR*, 7:793–800, 1934.

[26] K. Denker, B. Lehner, and G. Umlauf. Real-time triangulation of point streams. *Engineering with Computers*, 27:67–80, 2011.

[27] O. Devillers. On deletion in Delaunay triangulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 181–188, New York, NY, USA, 1999. ACM.

[28] O. Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13:163–180, 2002. special issue on triangulations.

[29] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *SCG '01: Proceedings of the seventeenth annual symposium on computational geometry*, pages 106–114, New York, NY, USA, 2001. ACM.

[30] O. Devillers and M. Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 313–319, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[31] R. Dubes and A. Jain. Clustering methodologies in exploratory data analysis. *Advances in Computers*, 19:113–228, 1980.

[32] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization'97*, pages 81–88. IEEE Computer Society Press Los Alamitos, CA, USA, 1997.

[33] B. Duran and P. Odell. Cluster analysis: A survey. *Lecture Notes in Economics and Mathematical Systems*, 1974.

[34] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2(1):137–151, 1987.

[35] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *J. of Res. the Nat. Bureau of Standards*, 69B:125–130, 1965.

[36] T. Fang and L. Piegl. Delaunay triangulation using a uniform grid. *IEEE Computer Graphics and Applications*, 13(3):36–47, 1993.

[37] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. *Advances in Knowledge Discovery and Data Mining*, pages 1–34, 1996.

[38] J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Algorithmica*, 41(1):25–41, 2004.

[39] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[40] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Towards an adaptive approach for mining data streams in resource constrained environments. *Data Warehousing and Knowledge Discovery*, pages 189–198, 2004.

[41] H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, 23(2):221–234, 1976.

[42] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[43] Georgia Institute of Technology, Large geometric models archive. `http://www.cc.gatech.edu/projects/large_models/index.html`.

[44] GEOVAP, spol. s r.o., Pardubice, Czech Republic. `www.geovap.cz`.

[45] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the International Conference on Very Large Data Bases*, pages 541–550, 2001.

[46] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th annual ACM symposium on Theory of computing*, pages 389–398. ACM New York, NY, USA, 2002.

[47] P. Goswami, F. Erol, R. Mukhi, R. Pajarola, and E. Gobbetti. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kD-trees. *The Visual Computer*, pages 1–15, 2012.

[48] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 58–66, New York, NY, USA, 2001. ACM.

[49] J. P. Grossman and W. J. Dally. Point sample rendering. In G. Drettakis and N. L. Max, editors, *Proceedings of Eurographics Workshop on Rendering*, pages 181–192. Springer, 1998.

[50] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 649–657, 1998.

[51] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, 2003.

[52] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[53] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[54] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1):381–413, 1992.

[55] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[56] J. Holenda. *O maticích*. Vydavatelský servis, Plzeň, 2007.

[57] H. Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[58] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[59] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[60] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. *Computer Graphics*, 27(Annual Conference Series):19–26, 1993.

[61] L. Hu, P. V. Sander, and H. Hoppe. Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 169–176, New York, NY, USA, 2009. ACM.

[62] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[63] A. Imiya and T. Sakai. Combinatorial properties of scale space singular points. *Combinatorial Image Analysis*, pages 333–346, 2006.

[64] P. Indyk. Algorithms for dynamic geometric problems over data streams. In *STOC '04: Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 373–380, New York, NY, USA, 2004. ACM.

[65] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM*, 53(3):307–323, 2006.

[66] M. Isenburg and P. Lindstrom. Streaming meshes. In *Visualization '05*, pages 231–238, 2005.

[67] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Shewchuk. Streaming compression of tetrahedral volume meshes. In *Proceedings of Graphics Interface 2006*, pages 115–121. Canadian Information Processing Society Toronto, Ontario, Canada, 2006.

[68] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. *IEEE Visualization, 2003*, pages 465–472, 2003.

[69] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *ACM SIGGRAPH 2005 Sketches*, page 136, New York, NY, USA, 2005. ACM.

[70] M. Isenburg, Y. Liu, J. R. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056, 2006.

[71] A. Jain and P. Flynn. Image segmentation using clustering. *Advances in Image Understanding: A Festschrift for Azriel Rosenfeld*, pages 65–83, 1996.

[72] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall advanced reference series. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[73] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[74] K. Jain and V. V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *IEEE Symposium on Foundations of Computer Science*, pages 2–13, 1999.

[75] V. Karamcheti, D. Geiger, Z. Kedem, and S. Muthukrishnan. Detecting malicious network traffic using inverse distributions of packet contents. In *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 165–170, New York, NY, USA, 2005. ACM.

[76] B. King. Step-wise clustering procedures. *Journal of the American Statistical Association*, 62(317):86–101, 1967.

[77] J. Kohout. Selected problems of parallel computer graphics. Technical report, University of West Bohemia, Univerzitní 22, Pilsen, 2004.

[78] J. Kohout. *Delaunay Triangulation in Parallel and Distributed Environment*. PhD thesis, University of West Bohemia, Pilsen, Czech Republic, 2005.

[79] Y.-M. Koo and B.-S. Shin. An efficient point rendering using octree and texture lookup. In *Computational Science and Its Applications — ICCSA 2005*, volume 3482 of *Lecture Notes in Computer Science*, pages 1187–1196. Springer, 2005.

[80] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *SODA: ACM-SIAM Symposium on Discrete algorithms*, pages 1–10, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[81] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[82] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. *SIGCOMM Computer Communication Review*, 35(4):217–228, 2005.

[83] C. Lawson. Software for c^1 surface interpolation. *Mathematical Software*, 3:161–194, 1977.

[84] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 259–266, Washington, DC, USA, 2002. IEEE Computer Society.

[85] M. Levoy and T. Whitted. The use of points as rendering primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill, 1985.

[86] J.-H. Lin and J. S. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44:245–249, 1992.

[87] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM.

[88] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proceedings of the conference on Visualization'01*, pages 363–371. IEEE Computer Society Washington, DC, USA, 2001.

[89] L. Lovász and M. D. Plummer. *Matching Theory*. Elsevier Science, North-Holland, Amsterdam, 1986.

[90] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics*, 31:199–208, 1997.

[91] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[92] M. O. M. de Berg, M. van Kreveld and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[93] J. B. Macqueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[94] P. Magillo. *Spatial Operations on Multiresolution Cell Complexes*. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, University of Genova, Italy, 1999. Report No. DISI-TH-1999-03.

[95] M. Mahdian, E. Markakis, A. Saberi, and V. Vazirani. A greedy facility location algorithm analyzed using dual fitting. *Lecture Notes in Computer Science*, 2129:127–137, 2001.

[96] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

[97] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008. Also available online at `http://informationretrieval.org/`.

[98] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, chapter 17 Hierarchical clustering. Cambridge University Press, July 2008. Also available online at `http://informationretrieval.org/`.

[99] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, chapter 16 Flat clustering. Cambridge University Press, July 2008. Also available online at `http://informationretrieval.org/`.

[100] The Grand Canyon data set. Data obtained from The United States Geological Survey (USGS), with processing by Chad McCabe of the Microsoft Geography Product Unit.

[101] D. H. McLain. Two dimensional interpolation from random data. *The Computer Journal*, 19(2):178–181, 1976.

[102] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[103] A. Meyerson. Online facility location. In *FOCS '01: IEEE Symposium on Foundations of Computer Science*, pages 426–431, Washington, DC, USA, 2001. IEEE Computer Society.

[104] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[105] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc, 2005. Also issued as *Foundations and trends in theoretical computer science*, 1(2), 2005. Manuscript available at `http://www.cs.rutgers.edu/~muthu/stream-1-1.ps`.

[106] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *20th International Conference on Very Large Data Bases*, pages 144–155, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[107] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. In *IEEE International Conference on Data Engineering*, pages 685–694, 2002.

[108] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.

[109] R. Pajarola. Efficient level-of-details for point based rendering. In *Computer Graphics and Imaging*, pages 141–146. IASTED/ACTA Press, 2003.

[110] R. Pajarola. Stream-processing points. In *IEEE Visualization*, pages 239–246. IEEE Computer Society, 2005.

[111] F. Preparata and M. Shamos. *Computational geometry: An introduction*. Springer, 1985.

[112] C. Prince. Progressive meshes for large models of arbitrary topology. Technical report, University of Washington, 2000.

[113] E. Puppo. Variable resolution terrain surfaces. In *Proceedings 8th Canadian Conference on Computational Geometry*, pages 202–210, 1996.

[114] E. Puppo. Variable resolution triangulations. *Computational Geometry: Theory and Applications*, 11(3–4):219–238, 1998.

[115] E. Rasmussen. Clustering algorithms. *Information retrieval: Data structures and algorithms*, pages 419–442, 1992.

[116] J. R. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Geometric Modelling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[117] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press / Addison-Wesley Publishing Co.

[118] M. Sainz and R. Pajarola. Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879, 2004.

[119] S. Schaefer and J. Warren. Adaptive vertex clustering using octrees. In *SIAM Geometric Design and Computing*, 2003.

[120] D. Schmalstieg and G. Schaufler. Smooth levels of detail. In *In Proc. of IEEE 1997 Virtual Reality Annual International Symposium*, pages 12–19. IEEE Computer Society Press, 1997.

[121] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

[122] D. B. Shmoys. Approximation algorithms for facility location problems. In *APPROX '00: Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *Lecture Notes in Computer Science*, pages 27–33, London, UK, 2000. Springer-Verlag.

[123] D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems (extended abstract). In *ACM Symposium on Theory of Computing*, pages 265–274, 1997.

[124] J. Skála. Algorithms for manipulation with large geometric and graphic data. Technical Report DCSE/TR-2009-02, Dept. of Computer Science and Engineering, University of West Bohemia, 2009.

[125] J. Skála. Data stream hierarchical clustering library, 2009. `http://www.kiv.zcu.cz/index.php?id=557&produkt_id=38`.

[126] J. Skála and I. Kolingerová. Clustering geometric data streams. In *SIGRAD 2007*, pages 17–23, 2007.

[127] J. Skála and I. Kolingerová. Dynamic hierarchical triangulation of a clustered data stream. *Computers & Geosciences*, 37(8):1092–1101, 2011.

[128] J. Skála and I. Kolingerová. Faster facility location and hierarchical clustering. *International Journal of Computers*, 5(1):132–139, 2011.

[129] J. Skála and I. Kolingerová. One-pass hierarchical clustering and dynamic 3D triangulation of huge streaming data, 2012. Submitted for publication in Graphical Models.

[130] J. Skála, I. Kolingerová, and J. Hyka. A Monte Carlo solution to the minimal Euclidean matching. In *ALGORITMY 2009*, pages 402–411, 2009.

[131] P. H. A. Sneath and R. R. Sokal. *Numerical taxonomy: The principles and practice of numerical classification.* W.H. Freeman, San Francisco, 1973.

[132] Stanford computer graphics laboratory. `http://graphics.stanford.edu/data/`.

[133] The Puget Sound data set. Data obtained from The United States Geological Survey (USGS), made available by The University of Washington.

[134] M. Vigo Anglada and N. Pla Garcia. Computing directional constrained Delaunay triangulations. *Computers & Graphics*, 24(2):181–190, 2000.

[135] The visible human project. `www.nlm.nih.gov/research/visible/`.

[136] G. Voronoi. Nouvelles applications des parametres continus a la theorie des formes quadratiques. *J. reine angew. Math*, 134:198–287, 1908.

[137] The walkthru project. www.cs.unc.edu/~walk/.

[138] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[139] WebGIS – Free terrain data. www.webgis.com/terr_world.html.

[140] K. Weiss and L. De Floriani. Sparse terrain pyramids. In *GIS '08: Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, New York, NY, USA, 2008. ACM.

[141] M. Wimmer and C. Scheiblauer. Instant points. In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006.

[142] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic: Behavior models and applications. *SIGCOMM Computer Communication Review*, 35(4):169–180, 2005.

[143] M. Zadravec and B. Žalik. An almost distribution-independent incremental Delaunay triangulation algorithm. *The Visual Computer*, 21(6):384–396, 2005.

[144] M. Zemek. A library for the construction of a dynamic regular and Delaunay tetrahedronization, 2009. http://www.kiv.zcu.cz/index.php?id=557&produkt_id=39.

[145] M. Zemek, J. Skála, I. Kolingerová, P. Medek, and J. Sochor. Fast method for computation of channels in dynamic proteins. In *Vision, Modeling, and Visualization 2008*, pages 333–342, 2008.

[146] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 103–114, 1996.